Examen de INF106

Documents et calculatrices **non autorisés**. Certaines signatures de fonctions sont **fournies en annexe**. Vous pouvez répondre sur le sujet: n'oubliez pas de le rendre **et** d'y inquer votre nom.

I. Questions de cours (7 points) – Une seule réponse possible pour les QCM

Question I.1 (1 point) – L'appel de la fonction execv...

a) crée un nouveau processus qui exécute le programme fourni en argument.
b) crée une copie du processus actuel.
c) remplace le programme exécuté par le processus par le programme fourni an argument

Question I.2 (1 point) – quels sont les 3 états possibles d'un processus? Pour vous aidez à identifier ces états, on dira par exemple qu'un processus passe de l'état ___ à l'état ___ lorsqu'il est choisi par l'ordonannceur. Un processus passe de l'état ___ à l'état ___ lorsqu'il exécute la fonction `scanf`.

Question I.3 (1 point) – Effectuez l'ordonnancement (sous forme de chronogramme) des trois processus A, B, C suivant l'algorithme **round robin (tourniquet)**. La durée d'un quantum est 1. Les processus ont une durée de 3 pour A, 2 pour B, et 1 pour C. Les trois processus démarrent à t=0, on les considère insérés dans la liste des processus prêts dans l'ordre alphabétique.

Question I.4 (1 point) – Pour partager des données entre processus, pourquoi utilise-t-on une variable stockée dans un fichier partagé plutôt qu'une variable en mémoire partagée ?

a) Parce que sans utilisation de bibliothèques particulières, deux processus UNIX ne partagent pas de données en mémoire, uniquement du code.
b) Parce que les services d'accès aux fichier (fopen, fscanf et fprintf) permettent d'utiliser des fonctions bloquantes qui rendent l'accès aux informations atomiques.
c) Pour illustrer les problèmes d'accès concurrents, qu'on ne peut pas observer avec des variables stockées en mémoire car l'accès aux variables est trop rapide (voire atomique).

Question I.5 (1 point) – Lorsque deux processus sont en interblocage, on observe:

a) que les deux processus utilise le processeur mais qu'aucun des deux ne progresse dans l'exécution de son code.
b) qu'aucun des processus n'utilise le processeur et qu'aucun des deux ne progresse dans l'exécution de son code.
c) que les deux processus utilise le processeur mais qu'un seul des deux progresse dans l'exécution de son code.

Question I.6 (1 point) – Qu'est-ce qu'un signal UNIX?

a) Un mécanisme intégré au processeur permettant d'interrompre l'exécution d'un processus lors de la survenue d'un événement
b) L'opération de l'ordonnanceur consistant à rendre actif un autre processus que celui qui était précédemment actif
c) Un mécanisme permettant au système d'exploitation de signaler à un processus la survenue d'un événement

Question I.7 (1 point) – Lorsqu'on exécute la commande "ls -l > ls_result",

a) On redirige le flux de sortie de la commande ls dans le fichier ls_result
b) On redirige le flux de sortie de la commande ls dans le flux d'entrée du programme ls_result
c) On ajoute la commande ls_result comme alternative équivalente à ls -l

II. Compréhension des services UNIX (7 points)

Question II.1 (2 points) – Un utilisateur de système UNIX souhaite écrire un programme qui exécute une fonction `path_enumeration()` dans N processus différents. Il demande de l'aide à deux élèves ingénieurs. Le code proposé par chaque élève est fourni ci-dessous.

Version 1

```
#include <unistd.h>
#define N 3

int main(int argc, char * argv[]) {
  int fork_ret = 0;
  for(int i=0; i<N-1; i++)
    fork_ret = fork();

  path_enumeration();

}
```

Version 2

```
#include <unistd.h>
#define N 3

int main(int argc, char * argv[]) {
  int fork_ret = 0;
  for(int i=0; i<N-1 && fork_ret==0; i++)
    fork_ret = fork();

  path_enumeration();

}
```

Quel version du programme vous semble correcte? Justifiez votre réponse en indiquant, pour chaque version, le nombre de processus qui exécutent la fonction `path_enumeration` (on inclura le processus qui exécute la fonction main initialement). On suppose N=3 comme dans le code ci-cessus.

Question II.2 (décomposée en questions II.2.a et II.2.b) – Vous avez cherché sur internet un code vous permettant de lire un fichier d'entrée et écrire dans un fichier de sortie. Vous trouvez le programme suivant:

```
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#define BUF_SIZE 5

int main(int argc, char * argv[]) {
  int fd_input = open("input_file", O_RDONLY);
  if(fd_input==-1)
    fprintf(stderr, "Error opening file\n");
  int fd_output = open("output_file", O_WRONLY|O_CREAT, 0666);
  size_t next_buf_size = 0;
  char buf[BUF_SIZE];
  do {
    next_buf_size = read(fd_input, buf, BUF_SIZE);
    if(next_buf_size == -1 ) {
      fprintf(stderr, "Error reading from file\n");
      return -1;
    }
    else {
      lseek(fd_output, 0, SEEK_SET);
      write(fd_output, buf, next_buf_size);
    }
  } while (next_buf_size==BUF_SIZE);
}
```

Question II.2.a (1 point) Quel intervalle de valeurs (entières) peut prendre la variable `next_buf_size` après l'appel à la fonction `read`?

Après exécution du programme sur un fichier dont le contenu est *Bonjour*, le contenu du fichier de sortie est *urnjo*. Si le contenu du fichier d'entrée est *Franchement*, le fichier de sortie contient *temen*. Si le contenu du fichier d'entrée est *Examen*, le fichier de sortie contient *nxame*.

Question II.2.b (2 points) En suivant les itérations succéssives de la boucle **do...while**, expliquez brièvement pourquoi l'exécution du programme sur un fichier dont le contenu est *Bonjour*, produit un fichier dont le contenu est *urnjo*. Que proposez vous pour résoudre ce problème et faire en sorte que le contenu du fichier de sortie soit le même que le contenu du fichier d'entrée?

Question II.3 (décomposée en questions II.3.a et II.3.b) - Nous considérons la technique de pagination présentée en cours sur une machine dont les adresses sont encodées sur 16 bits. On suppose que la taille d'une page est de 512 octets (précision: $512 = 2^9$). On considère un processus dont le code utilise 4 pages. On donne la table de pages suivante:

Numéro de page logique	Numéro de page physique
0	9
1	16
2	10
3	4

Question II.3.a (1 point) sur quelle adresse physique (en hexa-décimal) trouve-t-on l'instruction d'adresse (représentée en hexa-décimal): 0x020A

Question II.3.b (1 point) quelle est l'adresse logique (en hexa-décimal) de l'instruction qui se trouve à l'adresse physique (représentée en hexa-décimal): 0x1501

III. Utilisation de services UNIX (6 points)

Nous nous intéressons ici à l'implémentation d'une barrière. Le principe de la barrière est simple: les processus s'attendent mutuellement jusqu'à ce que le dernier processus arrive. Ce dernier processus libère tous les processus en attente, qui reprennent leur exécution.

Le canevas de code à compléter est fourni ci-dessous. Les commentaires dans le code identifient les parties à compléter, comme indiqué dans les questions ci-dessous. Dans ce code, nous supposons que le nombre de processus à synchroniser sur la barrière est donné par la macro `MAX_PROCESSES`. Le verrou `lock_for_shared_variables` sert à protéger l'accès aux variables partagées. Le nombre de processus ayant atteint la barrière est stocké dans un fichier partagé. Les fonctions `init_reached`, `incr_reached` et `decr_reached` ont pour effet d'initialiser, d'incrémenter et de décrémenter le nombre de processus qui ont atteint la barrière, et d'enregistrer le résultat dans le fichier partagé. La fonction `get_reached` lit ce fichier et retourne le nombre de processus qui ont atteint la barrière. Le code de ces fonctions est ignoré pour simplifier le code à compléter, nous précisons simplement que leur exécution n'est pas atomique.

Question III.1 (1 point) – Utilisez les fonctions `sem_post` et `sem_wait` pour protéger l'accès au fichier partagé qui comptabilise le nombre de processus ayant atteint la barrière. Les parties du code à compléter sontidentifiées avec le commentaire `Protect shared variables`.

Question III.2 (2 points) – Déclarez et initialisez le (ou les) sémaphore(s) permettant d'implémenter le mécanisme de la barrière, en utilisant la fonction `sem_open`. Utilisez la fonction `sem_wait` et le(s) sémaphore(s) de la barrière pour bloquer les processus qui arrivent sur la barrière. Utilisez la fonction `sem_post` pour libérer les processus en attente une fois qu'il y en a suffisament qui sont arrivés à la

barrière. Les parties du code à compléter sont identifiées avec le commentaire `Barrier implementation`.

Question III.3 (décomposée en questions III.3.a, III.3.b, et III.3.c) – Nous voulons borner le temps d'attente des processus à la barrière: un processus ne doit pas rester bloqué plus de 5 secondes. Utilisez le signal `SIGALRM` pour vous assurer qu'un processus est réveillé au bout de 5 secondes d'attente. La fonction `on_timeout` sera appelée sur échéance du délai de 5 secondes. Lorsque cette fonction est exécutée, le processus qui a reçu le signal se termine. Les parties de code à compléter sont identifiées avec le commentaire `Timeout implementation`.

Question III.3.a (1 point) – complétez le code pour que la fonction `on_timeout` soit appelée lorsqu'un processus a passé plus de 5 secondes à attendre l'ouverture de la barrière.

Question III.3.b (1 point) – on suppose un scénario dans lequel 3 processus ont atteint la barrière. Un processus a attendu plus de 5 secondes et s'est donc terminé en exécutant la fonction `on_timeout`. Puis 97 processus sont arrivés à la barrière. On note ici que MAX_PROCESSES vaut 100, donc la barrière se lève. A l'issue de l'exécution de la boucle for, ligne 54 à 63, quelle est la valeur du (ou des) compteur(s) associé(s) au(x) sémaphore(s) que vous avez utilisé(s) pour implémenter le mécanisme de la barrière? Est-ce que cela posera problème si on ré-exécute le programme?

Question III.3.c (1 point) – A la ligne 67 on propose d'utiliser la fonction `sem_getvalue`, pour connaitre la valeur du compteur du (ou des) sémaphore(s) que vous avez utilisé(s) pour implémenter le mécanisme de la barrière. Proposez un code qui s'appuie sur cette fonction pour que le code de la barrière puisse être exécuté plusieurs fois sans qu'il soit nécessaire de détruire le (ou les) sémaphore(s) que vous avez utilisé(s) pour implémenter le mécanisme de la barrière.

Indication sur la signature de `sem_getvalue`: le paramètre `sem` est le sémaphore, `sval` est l'adresse d'un entier qui contiendra la valeur du compteur si l'appel réussit.

```
int sem_getvalue(sem_t *sem, int *sval);
```

Note: appeler `sem_post` dans le handler de signal `on_timeout` n'est pas une option car cela pourrait libérer un autre processus que celui pour lequel le délai est arrivé à échéance...

```
1    #define MAX_PROCESSES 100
2
3    sem_t lock_for_shared_variables ;
4
5
6                                                  // Barrier implementation
7
8
9    int get_reached(); // returns the number of process at the barrier
10   void init_reached() ; // init (0) the number of process at the barrier
11   void incr_reached() ; // increment (+1) the number of process at the barrier
12   void decr_reached() ; // decrement ( 1) the number of process at the barrier
13   void child_function() ; // function executed by processes
14   void on_timeout (int sig_nb ) ; // signal handler (SIGALRM)
15
```

```
16  int etat;
17
18  int main (int argc , char * argv[]) {
19    init_reached();
20    /* Create and initialize semaphores */
21    lock_for_shared_variables = sem_open ( "/sem1" , O_CREAT, 0644, 1) ;
22
23                                                  // Barrier implementation
24
25
26    /** Creation of child processes **/
27    for ( i = 0 ; i < MAX_PROCESSES ; i ++) {
28      int f = fork () ;
29      if(f==0) {
30        child_function ( ) ;
31      } else {
32        break;
33      }
34    }
35
36    /* Wait for the end of all the processes */
37    while ( wait (& etat ) != -1)
38      printf ( " main - end of child process with state : %04x (hexa) \n" , etat ) ;
39    // Deletion of semaphores is ignored
40  }
41
42  void on_timeout (int sig_nb)
43  {
44    // Could implement a specific behavior (longjmp, etc). Ignored for now.
45    exit(-1);
46  }
47
48  void child_function() {
49
50                                                  // Protect shared variables
51    int reached_nb = get_reached() ;
52    if( reached_nb == MAX_PROCESSES-1) {
53      int i ;
54      for ( i =0; i <reached_nb ; i ++){
55        /* Release a process present at the barrier */
56
57
58                                                  // Barrier implementation
59
60
61        /* Decrement the number of processes at the barrier */
62        decr_reached () ;
63      }
64
65                                                  // Protect shared variables
66
67                                                  // Timeout implementation
68
69    }
70    else {
71      /* Increment the number of processes that reached the barrier */
72      incr_reached();
73      /* process should be wating at the barrier */
74
75
76                                                 // Timeout implementation
77
78
79                                                // Protect shared variables
80    }
81  }
```

Annexe

Voici, pour vous aider, quelques signatures de fonctions qui pourraient être utiles:

```
typedef unsigned long int size_t;
typedef long int ssize_t;
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, void *buf, size_t count);
typedef long int off_t;
off_t lseek(int fd, off_t offset, int whence);

int sem_post(sem_t *sem);
int sem_wait(sem_t *sem);
sem_t *sem_open(const char *name, int oflag);
int sem_close(sem_t *sem);
int sem_unlink(const char *name);

pid_t fork(void);
pid_t wait(int *wstatus);

int lockf(int fd, int cmd, off_t len);
int open(const char *pathname, int flags);

typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
unsigned int alarm(unsigned int seconds);
int kill(pid_t pid, int sig);
```