ROB4 - INFORMATIQUE SYSTEME

15 Novembre 2015
CORRECTION Devoir sur table
Documents papier autorisés. Ordinateurs, téléphones et calculatrices interdits.

NOM et PRÉNOM:

Barème :

Exercice	Points
1	9
2	1
3	1
4	1.5
5	1.5
6	3
7	3

Exercice 1 – QCM (9 points - 0.5 par question)

Cochez la bonne réponse (une seule par question).

1. Un processus *zombie* :

   o ne peut jamais être tué
   X disparaît quand son processus parent appelle `wait()`
   o ne peut être tué que par l'administrateur du système (root)
   o doit être éliminé rapidement car il peut facilement detruire un utilisateur du système
   o n'est plus visible dans la liste des processus
   o aucune des propositions précédentes n'est correcte

   Un processus devient zombie quand il est terminé mais que son parent ne l'a pas encore attendu avec `wait()` ou `waitpid()` ; il disparaît donc quand le parent utilise un ce ces appels.

2. Considérons `int main(int argc, char** argv)`. `argv[0]` :

- o est NULL s'il n'y a pas de paramètres
- X contient le nom du programme
- o contient le nombre d'arguments passés au programme
- o contient le premier argument passé au programme
- o est de type `char**`
- o aucune des propositions précédentes n'est correcte

> `argv[0]` contient le nom du programme et est de type `char*`. `argv[i]` pour $i > 0$ contient les différents arguments. `argc` contient le nombre d'éléments de `argv` (donc le nombre d'arguments +1).

3. L'appel `execv` :

- o Entraîne la création d'un nouveau processus
- o Entraîne la création d'un nouveau thread
- o Ne permet pas d'exécuter un programme avec des arguments
- o Renvoie 0 en cas de succès
- o Ne peut pas échouer
- X Aucune des propositions précédentes n'est correcte

> `int execv(const char *path, char *const argv[])` entraîne une mutation de processus : il permet d'exécuter le programme dont le chemin est fourni par `path` avec les arguments `argv`. Il ne crée pas de nouveau processus (le processus existant exécute le nouveau code), ni de thread, il peut échouer (par exemple si le programme n'est pas trouvé) et ne retourne jamais en cas de succès étant donné que le programme chargé par `execv` remplace le programme en cours d'exécution.

4. Le segment `.text` de l'espace d'adressage virtuel d'un processus :

- o Contient uniquement des chaînes de caractères
- o Contient les variables globales utilisées par le processus
- o Contient le code source C du programme
- X Est l'espace mémoire dans lequel pointent les pointeurs de fonction
- o Contient un texte décrivant le processus
- o Aucune des propositions précédentes n'est correcte

Le segment `.text` contient le programme compilé et assemblé (pas son code source C). Il contient donc les fonctions, et donc est l'espace dans lequel pointent les pointeurs de fonction.

5. Une erreur de segmentation (segmentation fault) :

- o Est uniquement obtenue en déréférencant le pointeur NULL
- o Entraîne toujours la fin immédiate du processus concerné
- o Transforme le processus concerné en zombie
- o Peut entraîner une corruption de la mémoire d'un autre processus
- X Entraîne l'envoi du signal SIGSEGV au processus
- o Aucune des propositions précédentes n'est correcte

Une erreur de segmentation est un plantage d'une application qui a tenté d'accéder à un emplacement mémoire qui ne lui était pas alloué. Elle est souvent causée par le déréférencement d'un pointeur invalide (de valeur NULL ou ayant une autre valeur invalide). Elle entraîne l'envoi du signal SIGSEGV au processus. Par défaut, celui-ci entraîne la terminaison du processus, mais ce n'est pas *toujours* le cas étant donné que ce signal peut être bloqué, ignoré ou redirigé pour avoir un autre effet (comme vous l'avez vu en TP).

6. Le fichier `main.c` contenant du code C valide, exécuter `gcc -c main.c` :

- X Réalise la compilation du fichier `main.c`
- o Réalise une édition de liens
- o Permet d'ajouter au programme des informations de débogage
- o Génère une erreur car il manque des arguments
- o Génère un fichier exécutable
- o Aucune des propositions précédentes n'est correcte

`gcc -c main.c` réalise la compilation (et l'assemblage) du code source dans `main.c`. Il génère un fichier objet `main.o`. Pour obtenir un fichier exécutable, on doit ensuite réaliser l'édition de liens sur ce fichier objet (et potentiellement d'autres), par exemple avec `gcc main.o -o monexe`. Pour ajouter des informations de débogage au programme, il faut utiliser l'argument supplémentaire `-g`

7. Le protocole UDP :

- o Ne permet d'envoyer des données qu'à un ordinateur se trouvant sur le même sous-réseau
- o Permet de garantir que les données arriveront au destinataire avant un délai donné
- o Est un protocole de routage
- o N'est pas utilisé sur Internet
- X Ne permet pas de garantir que les données arrivent dans le même ordre qu'elles ont été émises
- o Aucune des propositions précédentes n'est correcte

> Le protocole UDP est un protocole de transport (couche 4), fréquemment utilisé sur Internet tout comme le protocole TCP. C'est un protocole non connecté avec très peu de contrôle de flux, il ne permet pas de garantir que les données arrivent dans un ordre particulier, ni même qu'elles arrivent tout court. Il souvent est utilisé dans les cas ou acheminement rapide des données est important, mais il ne permet pas pour autant de garantir que les données arriveront avant un délai donné.

8. On souhaite réaliser avec gcc l'édition de liens d'un programme qui utilise la bibliothèque `libtoto.so`. Lors de l'appel à gcc, il faut ajouter l'argument :

- o `-libtoto`
- o `-llibtoto`
- X `-ltoto`
- o `-L libtoto.so`
- o Aucun argument particulier n'est nécessaire
- o Aucune des propositions précédentes n'est correcte

> Il faut utiliser `-l`, et la syntaxe correcte est `-ltoto`. `-L` sert a spécifier un répertoire ou chercher les bibliothèques.

9. Une situation d'interblocage (*deadlock*) se produit :

- o Quand deux threads essayent de vérouiller le même mutex
- X Quand deux threads attendent chacun qu'un mutex vérouillé par l'autre soit libéré
- o Quand deux threads essayent tous les deux de modifier la valeur d'une variable qui n'a pas été protégée par un mutex
- o Quand le processus principal se termine avant un thread qu'il a crée
- o Quand l'appel à `pthread_create` renvoie une erreur
- o Aucune des propositions précédentes n'est correcte

Il faut utiliser `-l`, et la syntaxe correcte est `-ltoto`. `-L` sert a spécifier un répertoire ou chercher les bibliothèques.

10. Laquelle de ces affirmations sur les interruptions est **fausse** ?

- o Les frappes sur le clavier provoquent des interruptions matérielles
- o Une interruption provoque l'exécution de code en mode noyau
- X Une interruption logicielle provoque la terminaison immédiate du processus en cours d'exécution
- o Une division par zéro peut provoquer une interruption logicielle
- o Les appels systèmes utilisent des interruptions logicielles
- o Aucune des affirmations précédentes n'est fausse

Les interruptions entraînent la suspension temporaire de l'exécution du code du processus et l'exécution du code du gestionnaire d'interruption correspondant, mais l'exécution du processus reprend ensuite. Elles ne causent généralement pas la terminaison du processus.

11. Etant donné le code suivant :

```
1  typedef unsigned char ui8_t; // 8 bits integer
2  typedef unsigned short int ui16_t; // 16 bits integer
3  typedef unsigned int ui32_t; // 32 bits integer
4
5  ui8_t a = 0xFF;
6  ui8_t b = 1 << 5;
7  ui16_t c = 0xFF00;
8
9  ui32_t temp = ((a << 8) | b) <<8;
10 ui32_t out = temp & c;
```

Indiquez laquelle de ces propositions logique est vraie après l'exécution :

- o `out == b | 0xFF00`
- o `out == 0`
- o `out == b « 16`
- X `out == 1 « 13`
- o `out == (1 « 21) | 0xFF00`
- o Toutes ces propositions sont fausses

Au départ, on a en notation binaire
— a == 0b11111111
— b == 0b00100000 (1 décalé de 5 positions vers la gauche)
— c == 0b1111111100000000
La ligne 8 :
— Décale `a` de 8 positions vers la droite : `a « 8 == 0b1111111100000000`
— Réalise un ou binaire avec `b` de 8 positions vers la droite :

`(a « 8) | b == 0b1111111100000000 | 0b00010000`
`(a « 8) | b == 0b1111111100010000`
— Décale encore le résultat de 8 positions vers la droite :
`temp == 0b11111111001000000000000`
La ligne 9 réalise un et binaire entre `temp` et `c` : on ne garde donc que les bits qui sont à 1 dans les deux cas, c'est à dire : `out == 0b00000000001000000000000 == 1«13`

12. Un processus ouvre en écriture (avec `open("myfifo",O_WRONLY)`) un fifo qui existe mais n'a pas été ouvert en lecture par un autre processus. Que se passe-t-il ?

- o L'appel à `open` échoue (il renvoie -1) et `errno` prend la valeur correspondant à l'erreur "Broken pipe"
- o Le processus reçoit le signal SIGPIPE
- o Rien de particulier, le fifo est ouvert en écriture et l'exécution continue
- X L'appel à `open` est bloquant jusqu'à ce que le fifo soit ouvert en lecture.
- o Le fifo est ouvert en lecture-écriture
- o Aucune des propositions précédentes n'est correcte

La situation décrite n'a rien de problématique, mais l'appel est bloquant jusqu'à ce que le fifo soit également ouvert en lecture (de façon à synchroniser les deux processus).

13. Les tubes anonymes (*pipes*) crées par la fonction `pipe()`.

- X Peuvent uniquement être utilisés pour communiquer entre des processus apparentés
- o Entraînent la création d'un fichier sur le disque dans `/tmp`
- o Doivent être ouverts avec `open()` avant d'être utilisés
- o Peuvent être fermés et rouverts plusieurs fois
- o Ne peuvent être utilisés que pour transférer des données binaires
- o Aucune des propositions précédentes n'est correcte

Les pipes sont qualifiés de tubes *anonymes* car ils ne sont pas associés à un fichier dans le système de fichier (un nom de fichier). Ils ne peuvent donc être utilisés que pour communiquer entre des processus apparentés, en exécutant `pipe` avant le fork. Ils sont définitivement perdus une fois qu'un des deux descripteurs du pipe a été fermé des deux côtés.

14. Le mécanisme de la copie en écriture (pour la gestion mémoire).

- X Permet d'économiser la mémoire physique lors de la création d'un processus
- o Empêche un processus de lire l'espace mémoire d'un autre processus
- o Permet de créer un espace mémoire partagé au sein duquel deux processus peuvent échanger des donnéees
- o Permet d'économiser l'espace disque lors de l'écriture de fichiers
- o Protège contre les erreurs de segmentation
- o Aucune des propositions précédentes n'est correcte

> La copie en écriture signifie qu'immédaietement après un `fork`, les processus parent et enfant vont avoir des espaces mémoire virtuels se référant largement aux mêmes pages de mémoire physique, une page de mémoire physique n'étant réellement dupliquée que quand l'un des deux processus écrit dans cette page. Il permet d'économiser la mémoire physique lors de la création d'un processus.

15. Le signal SIGKILL

- o Ne peut être envoyé que par l'utilisateur root
- o Peut être bloqué avec l'appel sigprocmask()
- X Entraîne la fin immédiate du processus cible
- o Termine les processus enfants du processus cible
- o Entraîne l'arrêt du système
- o Aucune des propositions précédentes n'est correcte

> Le signal SIGKILL, qui porte généralement le numéro 9, entraîne la fin immédiate du processus cible. C'est un des deux seuls signaux (avec SIGSTOP) à ne pas pouvoir être bloqué.

16. Une fonction a pour prototype : `void transformer_chaine(const char* in, char* out)`. Le qualificatif `const` dans `const char* in` signifie que :

- o Lors de l'appel de la fonction, le premier argument doit être une constante
- o Lors de l'appel de la fonction, le premier argument doit obligatoirement être une variable de type `const char *`
- X La fonction ne modifiera pas les données pointées par `in`
- o La fonction ne doit pas être appelée dans un thread
- o La mémoire pointée par `in` ne doit pas avoir été alloué par `malloc()`
- o Aucune des propositions précédentes n'est correcte

Le qualifieur `const` dans la déclaration d'un argument de type pointeur indique que la fonction ne modifiera pas les données pointées par le pointeur (elle se contentera de les lire). Une tentative de modifier ces données dans le code de la fonction entraînera une erreur de compilation.

17. Etant donné :

```
1 char * c = malloc(42*sizeof(char));
```

Indiquez laquelle de ces propositions logiques est vraie :

- o `sizeof(c) == sizeof(char)`
- X `sizeof(c) == sizeof(char*)`
- o `sizeof(c) == 42*sizeof(char)`
- o `sizeof(c) == 42*sizeof(char*)`
- o `sizeof(c) == 0`
- o Aucune des propositions précédentes n'est correcte

Indépendemment de sa valeur, `c` est une variable de type `char*`, et sa taille est donc celle de ce type (à savoir généralement 8 octets sur une machine 64 bits).

18. Etant donné :

```
1  int * foo(int* x)
2  {
3          int out = *x + 23;
4          return &out;
5  }
6
7  int main(int argc, char* argv[])
8  {
9    int i = 42;
10   int * outptr;
11   outptr = foo(&i);
12   printf("Out = %d\n",*outptr);
13   return 0;
14 }
```

Indiquez laquelle de ces propositions est vraie :

- o La mémoire associée à la variable `i` est allouée sur le tas (*heap*)
- o La mémoire associée à la variable `i` est dans la tranche de pile (*stack frame*) correspondant à `foo`.
- o Ce code risque de générer une erreur de segmentation (ou autre erreur mémoire) car `foo` ne peut pas accéder à `i`.
- X Ce code risque de générer une erreur

de segmentation (ou autre erreur mémoire) car `foo` renvoie l'adresse d'une variable locale.

o Ce code risque de générer une erreur de segmentation (ou autre erreur mémoire) car le pointeur `outptr` n'est pas initialisé à NULL.

o Aucune des propositions précédentes n'est correcte

o La mémoire associée à la variable `i` est sur la pile, dans la tranche de pile (*stack frame*) de `main`. Lorsque `foo` est appelée par `main`, elle peut donc sans problème accéder à cette variable par un pointeur. En revanche, le pointeur vers `out` renvoyé par `foo` est invalide dans `main`, étant donné qu'il pointe vers une adresse dans la tranche de pile de `foo`, qui cesse d'exister après l'exécution de cette fonction.

Exercice 2 (1 point)

1) (0.5 point) Ecrire le code de la fonction `swap` qui permute deux entiers (après l'appel, la valeur pointée par `x` doit être égale à celle initialement pointée par `y` et réciproquement) :

```
1 void swap(int *x, int *y)
2 {
3         int temp = *y;
4         *y = *x;
5         *x = temp;
6 }
```

2) (0.5 point) Si dans un programme nous avons declaré le suivant :
`int a=3, b=5;`
Comment appelle-t-on la fonction `swap` pour permuter les valeurs de a et b ?

```
swap(&a, &b);
```

Exercice 3 (1 point)

Ecrire la sortie et l'arbre des processus du programme suivant, appelé `fork1`. Vous pouvez choisir le PID du premier processus comme vous le souhaitez, et on considère pour cet exercice que ce programme est le seul s'exécutant à cet instant et que les PIDs sont assignés normalement, de facon séquentielle.

```c
#include <unistd.h>     // fork(),getpid()
#include <stdio.h>      // printf()

int main(void)
{
        int i;
        fork();
        for(i=0; i< 2; i++)
        {
                fork();
        }
        printf("process pid=%d ppid= %d \n",getpid(),getppid());
        while(1);
        return 0;
}
```

1) Sortie (0.5 point) :

```
process pid=7939 ppid= 7135
process pid=7942 ppid= 7939
process pid=7941 ppid= 7939
process pid=7944 ppid= 7941
process pid=7940 ppid= 7939
process pid=7945 ppid= 7940
process pid=7943 ppid= 7940
process pid=7946 ppid= 7943
```

2) Arbre des processus (0.5 point) :

```
fork(7939)-+-fork(7940)-+-fork(7943)---fork(7946)
           |            +-fork(7945)
           +-fork(7941)---fork(7944)
           +-fork(7942)
```

Exercice 4 (1.5 points)

Après la portion de code ci-dessous, spécifiez le contenu de la mémoire (1ère colonne ; 0.5 point) ainsi que ce que donnerait un `printf` de certaines expressions (2nde colonne ; 1 point) : valeurs ou adresses.

Si la valeur de certains espaces mémoires est indéterminées, laissez la case vide.

Si certains `printf` donneraient une erreur (à la compilation ou l'exécution), inscrivez "Erreur" dnas la case correspondante.

```
1  int b = 7;
2  int v[] = { 9, 8, 6};
```

0x..1	7	← b
0x..2	9	← v
0x..3	8	
0x..4	6	
0x..5		
0x..6		

`printf("...", ...);`

`b`	7	`&b`	0x..1
`*b`	Error	`&(b+1)`	Error
`*v + 1`	10	`&b+1`	0x..2
`*(v + 1)`	8	`++b`	8
`&(&v)`	Error	`**b`	Error

Exercice 5 (1.5 points)

On rappelle que la suite de Fibonacci est la suite F définie par $F_0 = 0$, $F_1 = 1$, et $\forall i \geq 2$, $F_i = F_{i-1} + F_{i-2}$. Il s'agit d'une suite d'entiers, strictement croissante pour $i \geq 2$, et qui croît pour atteindre rapidement de grandes valeurs (par exemple $F(100) \approx 3.5 \times 10^{20}$).

Le programme `fibonacci` a pour but de calculer les n premiers termes de la suite de Fibonacci, puis de les afficher, la valeur n étant passée en argument. La sortie attendue est par exemple :

```
% ./fibonacci 8
F(0) = 0
F(1) = 1
F(2) = 1
F(3) = 2
F(4) = 3
F(5) = 5
F(6) = 8
F(7) = 13
F(8) = 21
%
```

Voici le listing de `fibonacci.c` :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  // Compute the Fibonacci sequence up to value F(to_n)
5  // Stores the values in the array pointed by dest
6  void fibonacci(int to_n, int* dest)
7  {
8          dest[0] = 0;
9          dest[1] = 1;
10         for(int i = 2 ; i < to_n ; i++)
11                 dest[i] = dest[i-1]+dest[i-2];
12 }
13
14
15
16 int main(int argc, char* argv[])
17 {
18         if(argc != 2)
19         {
20                 printf("Usage: ./fibonacci <max n>\n");
21                 return 1;
22         }
23         int n = atoi(argv[1]);
```

```
24          if(n < 2)
25          {
26                  printf("n must be 2 or greater\n");
27                  return 1;
28          }
29          int * suite = NULL;
30          fibonacci(n, suite);
31          for(int i = 0 ; i <= n ; i++)
32                  printf("F(%d) = %d\n",i,suite[i]);
33
34          return 0;
35  }
```

1) (0.25 point) Le programme listé ci-dessus peut être compilé, mais son exécution provoque une erreur de segmentation. Pourquoi ?

Ligne 29, le pointeur `suite` est initialisé à NULL, puis immédiatement donné en argument à la fonction `fibonacci`, qui le déréférence.

2) (0.5 point) Comment peut-on corriger cette erreur ? Indiquez quelle(s) ligne(s) de code il convient de modifier ou d'ajouter, et quelles sont les modifications à effectuer.

Il aurait faut allouer l'espace nécessaire pour stocker `n+1` valeurs entières (valeurs de la suite de Fibonacci de 0 à n), en remplaçant la ligne 29 par :

```
1  int * suite = malloc((n+1)*sizeof(int));
```

(Il est bien sûr également possible d'utiliser `calloc`).) Il faut donc également penser à libérer la mémoire à la fin du programme en ajoutant ligne 34 :

```
1  free(suite);
```

3) (0.25 point) En plus de l'erreur de segmentation, ce programme comporte une autre erreur qui ne génère pas de plantage mais entraîne un comportement non conforme à la sortie demandée. Quelle est-elle et comment la corriger ?

Ligne 10, la condition d'arrêt de la boucle utilise une comparaison stricte, ce qui fait que le dernier terme calculé est celui d'indice n-1, or l'énoncé de la question demande d'aller jusqu'à la valeur n entrée.

Il faut donc remplacer la ligne 10 par :

```
1 for(int i = 2; i <= to_n ; i++)
```

Pour que cela fonctionne sans déclencher d'erreur de segmentation, il faut bien avoir pensé à allouer un tableau de taille n+1 à la question précédente.

Une fois ces deux erreurs corrigées, le programme fonctionne correctement pour de petites valeurs de n, cependant on constate à partir d'un certain indice (ici n=47) les résultats sont incorrects :

```
% ./fibonacci 50
F(0) = 0
F(1) = 1
F(2) = 1
F(3) = 2
F(4) = 3
F(5) = 5
...
F(43) = 433494437
F(44) = 701408733
F(45) = 1134903170
F(46) = 1836311903
F(47) = -1323752223
F(48) = 512559680
F(49) = -811192543
F(50) = -298632863
%
```

4) (0.5 point) Quelle est l'origine de ce comportement ? Comment peut-on corriger partiellement ce problème pour qu'il ne se pose que pour des valeurs de n beaucoup plus élevées ?

La suite de Fibonacci croissant rapidement, ses valeurs atteignent rapidement la valeur maximale stockable dans un `int`, généralement stocké sur 32 bits. Le type `int` étant signé, cette saturation provoque l'interprétation du résultat comme un nombre négatif.

Une solution serait de remplacer (dans les instructions, les déclarations

de fonction et le malloc) le type `int` par un `long` ou mieux un `unsigned long`, qui sont toujours stockés sur 64 bits sur une machine 64 bits.

Exercice 6 (3 points)

Voici le listing du programme `pipefork.c`. Les lignes 9, 15, 21, 25 et 39 contiennent des trous, où une partie du code a été remplacé par un commentaire.

```
1  #include <unistd.h> // fork, dup2, pipe
2  #include <stdio.h> // printf
3
4  #define BUF_SIZE 64
5
6  int main(int argc, char* argv[])
7  {
8          // Pipe creation
9          /* [RESULTAT QUESTION 2] */
10         pipe(pipefd);
11
12         // Fork
13         int pid = fork();
14
15         if(/* [RESULTAT QUESTION 3] */)
16         {
17                 // We are in the child
18                 close(pipefd[0]);
19
20                 // Redirect output to pipe
21                 dup2(/* [RESULTAT QUESTION 4] */)
22
23                 // Execute ls -l
24                 char* args[] = {"ls", "-l", NULL};
25                 /* [RESULTAT QUESTION 5] */ // never returns
26         } else {
27
28                 // We are in the parent
29                 close(pipefd[1]);
30
31                 char mybuf[BUF_SIZE]; // Read buffer
32                 int total = 0; // Total number of characters read
33                 int = 0; // Number of characters read at once
34
35                 /* Read the output of the pipe BUF_SIZE characters
36                    at a time until nothing more is read */
37                 do
38                 {
39                         // Read BUF_SIZE characters (at most)
40                         n = /* [RESULTAT QUESTION 6] */;
41                         total += n; // Add number read to total
```

```
42                  } while(n > 0);
43
44                  // Print total number of characters read
45                  printf("The child output %d characters.\n", total);
46                  close(pipefd[0]);
47          }
48          return 0;
49  }
```

1) (0.5 point) Malgré les trous, la structure et les commentaires du code permettent d'en comprendre le fonctionnement. Que fait ce programme ?

Il crée un processus fils qui exécute `ls -l`, redirige sa sortie vers un tube, et le processus parent lit le contenu du tube et compte les caractères lus jusqu'à ce qu'il n'y ait plus rien à lire, puis affiche ce nombre.

Complétez le code en remplaçant les 5 commentaires aux lignes 9, 15, 21, 25 et 39 (la réponse est, au plus, une seule ligne de code) :

2) (0.5 point) (l.9)

```
int pipefd[2];
```

3) (0.5 point) (l.15)

```
pid == 0
```

4) (0.5 point) (l.21)

```
pipefd[1],STDOUT_FILENO;
```

5) (0.5 point) (l.25)

```
execvp("ls",args);
```

6) (0.5 point) (l.40)

```
read(pipefd[0], mybuf, 64)
```

Exercice 7 (3 points)

Voici le listing du programme `consoprod.c`. Les lignes 25, 63, 73-75 et 77 contiennent des trous, où une partie du code a été remplacé par un commentaire. De plus les lignes mettant en œuvre les mutex ont elles aussi été supprimées. Il vous est demandé de donner le code permettant de rendre le programme fonctionnel :

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #include <unistd.h>
4  #include <stdlib.h>
5
6  #define NB_TRAVAILLEUR 4
7
8  typedef struct work {  // Parametres d'un travailleur:
9      int quantite_travail; // quantite de travail consomme
10     int temps_sommeil; // temps de repos apres travail
11     int num_travailleur; // numero du travailleur
12 } work;
13
14  /* Variable globale representant la quantite de travail a
15     effectuer */
16 int travail;
17
18 // Mutex protegeant la variable travail
19 pthread_mutex_t mutex;
20
21 /* Les threads "travailleur" consomment du travail, se reposent
22    et recommencent */
23 void* travailleur(void* arg) {
24     work local;
25     /* [RESULTAT QUESTION 4] */
26     printf("Bonjour, je suis le travailleur %d\n", local.num_travailleur);
27     while(1) {
28         if (travail > 0)
29         {
30             // Consommation du travail
31             travail -= local.quantite_travail;
32             if(travail < 0)
33                     travail = 0;
34             printf("Le travailleur %d a consomme du travail, il reste %d\n",
35                     local.num_travailleur, travail);
36         }
37         else {
38             // Aucun travail a consommer:
39             printf("POLLING\n");
```

```
40          }
41          // Repos
42          sleep(local.temps_sommeil);
43      }
44      pthread_exit(NULL);
45  }
46
47  // Le thread "patron" produit du travail regulierement
48  void* patron(void* arg) {
49      printf("Bonjour, je suis le patron\n");
50      while(1) {
51          travail += 100;
52          printf("Travail produit par le patron : %d\n", travail);
53          sleep(6);
54      }
55      pthread_exit(NULL);
56  }
57
58  int main() {
59      pthread_t patron_id;
60      pthread_t travailleur_id[NB_TRAVAILLEUR];
61
62      travail = 0;
63      /* [RESULTAT QUESTION 1] */
64      //Parametres des threads travailleurs
65      work travail_param[NB_TRAVAILLEUR];
66
67      // Creation du thread patron
68      pthread_create(&patron_id, NULL, patron, NULL);
69
70      // Creation des threads travailleurs
71      for (int i = 0; i < NB_TRAVAILLEUR; i++) {
72
73          /*                          */
74          /* [RESULTAT QUESTION 2] */
75          /*                          */
76
77          /* [RESULTAT QUESTION 3] */
78      }
79      while(1);
80  }
```

1) (0.5 point) Initialisez le mutex (ligne 63).

```
pthread_mutex_init(&mutex, NULL);
```

2) (0.5 point) Initialisez les champs de la structure avec des valeurs de votre choix dépendant de `i` (lignes 73-75).

Plusieurs possibilités sont bien sûr possibles, en voici une :

```
1 travail_param[i].quantite_travail = (1 + i) * 10;
2 travail_param[i].temps_sommeil = 1 + i;
3 travail_param[i].num_travailleur = i;
```

3) (0.5 point) Créez les threads travailleurs (ligne 77).

```
pthread_create(&travailleur_id[i], NULL, travailleur,
(void*)&travail_param[i]);
```

4) (0.5 point) Initialisez la structure `local` dans la fonction `travailleur` (ligne 25).

```
local = *((work*)arg)
```

5) (1 point) Donnez les lignes de code permettant l'acquisition et le relâchement du mutex et indiquez les lignes où placer l'un et l'autre et la position dans la ligne (par exemple "après la ligne 42").

On peut vérouiller le mutex avec `pthread_mutex_lock(&mutex);` et le relâcher avec `pthread_mutex_unlock(&mutex);`.

Il faut vérouiller le mutex avant de lire ou modifier la variable `travail` : avant la ligne 28 dans le travailleur et avant la ligne 51 dans le patron.

Il faut dévrouiller le plus tôt possible quand on ne touche plus à la variable `travail` : avant la ligne 42 dans le travailleur et avant la ligne 53 dans le patron. Il est particulièrement important de dévérouiller le mutex avant

- d'appeler `sleep` pour ne pas bloquer inutilement la variable pendant le repos.