Commencé le	vendredi 1 mars 2024, 18:00
État	Terminé
Terminé le	vendredi 1 mars 2024, 19:33
Temps mis	1 heure 33 min
Note	**13,55** sur 20,00 (**67,75**%)

Question 1

Terminé

Non noté

Directives :

1. Cet examen est composé de 14 questions pour une durée totale de 2 heures.
2. Pondération 30%.
3. En guise de documentation, vous aurez accès à votre compte Moodle et, par conséquent, à la totalité du site du cours INF2610, incluant les diapos, etc. Aucune autre documentation ne sera permise. Vous aurez droit à la calculatrice non-programmable. Nous vous fournirons également une feuille brouillon.
4. Aucune réponse aux questions durant l'examen.
5. **Sauf mention contraire, tous les appels système utilisés dans les questions sont supposés exempts d'erreurs et considèrent leurs options par défaut**.
6. Il n'est pas demandé de traiter les cas d'erreurs, ni d'inclure les directives d'inclusion dans les codes à compléter.
7. Pour les questions à choix multiples, **vous devez sélectionner une seule réponse**.
8. Il n'est pas possible de joindre des fichiers à vos réponses.
9. Lisez au complet chaque question avant d'y répondre.
10. Vous pouvez inclure vos commentaires au responsable du cours dans l'espace ci-après.

Bon examen à tous

Du coup, quelqu'un sait c'est quoi le STEP?

Question 2

Terminé

Non noté

Sur mon honneur, j’affirme que je compléterai cet examen en vertu du code de conduite de l'étudiant de Polytechnique Montréal et de sa politique sur le plagiat. J’affirme également que je compléterai cet examen par moi-même, sans communication avec personne, et selon les directives diffusées sur les canaux de communication.

Écrivez votre nom complet ainsi que votre matricule en guise d’approbation dans la zone de texte ci-dessous.

Omar Benzekri

2244082

Question 3

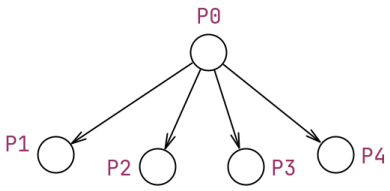
Correct

Note de 2,00 sur 2,00

Complétez le code ci-après pour créer l'arbre de processus décrit par la figure suivante (où P0 est le processus principal). Chaque processus parent doit attendre la fin de ses processus fils juste avant de se terminer. Chaque processus sans enfant doit se transformer pour exécuter la commande : echo <NOM> où <NOM> est le nom du processus (P1 à P4). Vous devez respecter l'ordre numérique de création des processus, e.g. le processus P1 doit être créé avant le processus P2, etc.. **Rappel : les appels système fonctionnent et ne retournent aucune erreur.**

int main() {
/*0*/
/*1*/
/*2*/
/*3*/
/*4*/
/*5*/
/*6*/
/*7*/
/*8*/
/*9*/
/*10*/
/*11*/
/*12*/
/*13*/ _exit(0); <-- *L'instruction /*13*/ est nécessairement **_exit(0);** et vous n'avez pas à l'insérer dans votre solution.*

}



Pour répondre à cette question, sélectionnez les instructions à insérer aux bons endroits.

/*0*/	if (fork()==0) { ✓
/*1*/	execlp("echo", "echo", "P1",NULL); ✓
/*2*/	} ✓
/*3*/	if (fork()==0) { ✓
/*4*/	execlp("echo", "echo", "P2",NULL); ✓
/*5*/	} ✓
/*6*/	if (fork()==0) { ✓
/*7*/	execlp("echo", "echo", "P3",NULL); ✓
/*8*/	} ✓
/*9*/	if (fork()==0) { ✓

/*10*/ execlp("echo", "echo", "P4",NULL); ✓

/*11*/ } ✓

/*12*/ while(wait(NULL)>0); ✓

Votre réponse est correcte.

La réponse correcte est :

/*0*/ → if (fork()==0) {,

/*1*/ → execlp("echo", "echo", "P1",NULL);,

/*2*/ → },

/*3*/ → if (fork()==0) {,

/*4*/ → execlp("echo", "echo", "P2",NULL);,

/*5*/ → },

/*6*/ → if (fork()==0) {,

/*7*/ → execlp("echo", "echo", "P3",NULL);,

/*8*/ → },

/*9*/ → if (fork()==0) {,

/*10*/ → execlp("echo", "echo", "P4",NULL);,

/*11*/ → },

/*12*/ → while(wait(NULL)>0);

Question 4

Correct

Note de 1,00 sur 1,00

Supposons un système monoprocesseur multiprogrammé où plusieurs processus sont en attente d'exécution et le processus en cours (que nous appellerons processus Z) exécute le programme suivant:

```
int main()
{
  for(int i=0; i<10; i = (i+1) % 10);
  return 0;
}
```

Complétez les phrases suivantes:

- Les processus en attente [ne pourront pas] ✓ s'exécuter, si le mode d'exploitation est de type "traitement par lots". [L'exécution du programme va monopoliser le processeur, et aucune gestion des priorités n'est possible avec ce mode d'exploitation.] ✓ .
- Les processus en attente [pourront] ✓ s'exécuter, si le mode d'exploitation est de type "temps partagé". [L'exécution du programme sera suspendue au bout de son quota de temps.] ✓ .
- Les processus en attente [ne pourront pas] ✓ s'exécuter, si le processus Z est le plus prioritaire et le mode d'exploitation est basé sur des priorités fixes. [Mais un processus de priorité supérieure à celle du processus Z pourrait s'exécuter.] ✓ .

ne pourront pas | pourront

L'exécution du programme va monopoliser le processeur, et aucune gestion des priorités n'est possible avec ce mode d'exploitation.

L'exécution du programme sera suspendue au bout de son quota de temps.

L'exécution du programme prendra fin au bout de son quota de temps.

Mais un processus de priorité inférieure à celle du processus Z pourrait s'exécuter.

Mais un processus de priorité supérieure à celle du processus Z pourrait s'exécuter.

Mais la priorité du processus Z redescendra automatiquement au bout de son quota de temps.

Votre réponse est correcte.

La réponse correcte est :
Supposons un système monoprocesseur multiprogrammé où plusieurs processus sont en attente d'exécution et le processus en cours (que nous appellerons processus Z) exécute le programme suivant:

```
int main()
{
  for(int i=0; i<10; i = (i+1) % 10);
  return 0;
}
```

Complétez les phrases suivantes:

- Les processus en attente [ne pourront pas] s'exécuter, si le mode d'exploitation est de type "traitement par lots". [L'exécution du programme va monopoliser le processeur, et aucune gestion des priorités n'est possible avec ce mode d'exploitation.].
- Les processus en attente [pourront] s'exécuter, si le mode d'exploitation est de type "temps partagé". [L'exécution du programme sera suspendue au bout de son quota de temps.].
- Les processus en attente [ne pourront pas] s'exécuter, si le processus Z est le plus prioritaire et le mode d'exploitation est basé sur des priorités fixes. [Mais un processus de priorité supérieure à celle du processus Z pourrait s'exécuter.].

Question 5

Incorrect

Note de 0,00 sur 2,00

Considérez le programme BOUCLE_D'OR décrit ci-après.

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

void main(int argc, char * argv[])
{
  int i;
  pid_t p;
  int n;

  if(argc != 2) {
    fprintf(stderr, "Veuillez s.v.p. fournir un argument et un seul.\n");
    exit(1);
  }

  n = atoi(argv[1]);
  p = 0;

  for(i = 1; i<n; ++i) {
    p = fork();
    if(p)
      break;
  }

  fprintf(stderr,
          "i=%d; ID processus = %ld; ID parent = %ld; ID enfant = %ld\n",
          i, (long)getpid(), (long)getppid(), (long)p);
  exit(0);
}
```

Vous lancez ce programme à la console avec comme seul argument la valeur 5. Le programme s'exécute correctement, et tous les appels systèmes fonctionnent sans erreur.

Une seule des sorties suivantes est possible. Laquelle?

Indice: l'exécution de la commande *ps* avant le lancement du programme donne le résultat suivant:

```
  PID TTY          TIME CMD
 4544 pts/2    00:00:00 bash
27176 pts/2    00:00:00 ps
```

-
```
i=1; ID processus = 48910; ID parent = 4544; ID enfant = 48912
i=2; ID processus = 48912; ID parent = 4544; ID enfant = 48913
i=3; ID processus = 48913; ID parent = 4544; ID enfant = 48914
i=4; ID processus = 48914; ID parent = 2765; ID enfant = 48915
i=5; ID processus = 48915; ID parent = 2765; ID enfant = 0
```
✗

```
i=1; ID processus = 18325; ID parent = 4544; ID enfant = 18326
i=2; ID processus = 18326; ID parent = 2765; ID enfant = 18327
i=3; ID processus = 18327; ID parent = 18326; ID enfant = 18328
i=4; ID processus = 18328; ID parent = 2765; ID enfant = 18330
i=5; ID processus = 18330; ID parent = 2765; ID enfant = 18331
i=6; ID processus = 18331; ID parent = 18330; ID enfant = 0
```

```
i=1; ID processus = 29857; ID parent = 4544; ID enfant = 29859
i=2; ID processus = 29859; ID parent = 2765; ID enfant = 29860
i=3; ID processus = 29860; ID parent = 4544; ID enfant = 29861
i=4; ID processus = 29861; ID parent = 2765; ID enfant = 29863
i=5; ID processus = 29863; ID parent = 2765; ID enfant = 29864
i=6; ID processus = 29864; ID parent = 4544; ID enfant = 0
```

```
i=1; ID processus = 30994; ID parent = 4544; ID enfant = 30995
i=2; ID processus = 30995; ID parent = 2765; ID enfant = 30996
i=3; ID processus = 30996; ID parent = 30995; ID enfant = 30997
i=4; ID processus = 30997; ID parent = 2765; ID enfant = 30999
i=5; ID processus = 30999; ID parent = 30997; ID enfant = 0
```

Votre réponse est incorrecte.

La réponse correcte est :

```
i=1; ID processus = 30994; ID parent = 4544; ID enfant = 30995
i=2; ID processus = 30995; ID parent = 2765; ID enfant = 30996
i=3; ID processus = 30996; ID parent = 30995; ID enfant = 30997
i=4; ID processus = 30997; ID parent = 2765; ID enfant = 30999
i=5; ID processus = 30999; ID parent = 30997; ID enfant = 0
```

Question 6

Incorrect

Note de 0,00 sur 1,50

Considérez le programme BOUCLE_D'OR de la question précédente.

À la lumière des informations fournies, lequel des énoncés suivants est manifestement *faux*?

- C'est le processus *systemd*, et non pas le processus *init*, qui adopte les processus orphelins sous cette implémentation de Linux.
- Sachant que la valeur du paramètre est de 5, il y aura exactement une valeur de "ID enfant" qui soit égale à zéro.
- Certains parents peuvent se terminer avant leur(s) enfant(s). ✕
- Le processus *systemd* porte le numéro 2765.
- La structure du programme assure que l'ordre numérique des "i" 1,2,3... sera respecté.

Votre réponse est incorrecte.

La réponse correcte est :
La structure du programme assure que l'ordre numérique des "i" 1,2,3... sera respecté.

Question 7

Correct

Note de 1,00 sur 1,00

Lequel des énoncés suivants est *vrai*?

- En mode d'annulation asynchrone, le thread ciblé par une requête d'annulation vérifie périodiquement s'il doit se terminer et ce, afin de permettre une sortie élégante.
- La concurrence implicite est une stratégie qui vise à transférer la création et la gestion des threads vers le matériel.
- Lorsqu'un thread appelle *execl()*, l'image complète du processus, et non pas seulement les structures du thread appelant, est remplacée par une nouvelle image. ✓
- Lorsqu'un thread appelle *fork()*, tous les threads du processus sont copiés, pas seulement le thread appelant.
- Un avantage de la bibliothèque *pthread* sur la bibliothèque *pth* est que *pthread* permet d'implémenter des threads non-préemptifs.

Votre réponse est correcte.

La réponse correcte est : Lorsqu'un thread appelle *execl()*, l'image complète du processus, et non pas seulement les structures du thread appelant, est remplacée par une nouvelle image.

Question 8

Partiellement correct

Note de 1,05 sur 1,50

Lesquelles des ressources suivantes sont partagées par les threads d'un même processus? Lesquelles sont partagées par un processus parent et son enfant?

* Ressource *	* Partagée entre threads d'un même processus? *	* Partagée entre un processus parent et son enfant? *
Registres	✓	✗
Segment de pile	✓	✗
Segment de données	✓	✓
Table des descripteurs de fichiers	✓	✓
Segments alloués par "mmap" avec l'option MAP_SHARED	✓	✗

OUI NON

Votre réponse est partiellement correcte.

Vous en avez sélectionné correctement 7.
La réponse correcte est :
Lesquelles des ressources suivantes sont partagées par les threads d'un même processus? Lesquelles sont partagées par un processus parent et son enfant?

* Ressource *	* Partagée entre threads d'un même processus? *	* Partagée entre un processus parent et son enfant? *
Registres	[NON]	[NON]
Segment de pile	[NON]	[NON]
Segment de données	[OUI]	[NON]
Table des descripteurs de fichiers	[OUI]	[NON]
Segments alloués par "mmap" avec l'option MAP_SHARED	[OUI]	[OUI]

Question 9

Incorrect

Note de 0,00 sur 1,00

Considérez le programme suivant que nous appellerons POMME_D'API. Ce programme fonctionne tel quel sans interblocage.

```
#include <unistd.h>
#include <sys/wait.h>

int main () {
  int fd[2]; pipe(fd);
  if(fork()==0)  {  // premier fils
    dup2(fd[0],0);
    close(fd[1]); // INSTRUCTION A
    close(fd[0]);
    char c;
    while (read(0,&c,1) >0) // INSTRUCTION R
      write(1,&c, 1); // INSTRUCTION B
    _exit(0);
  }
  if(fork()==0)  {  // second fils
    dup2(fd[1],1);
    close(fd[1]); // INSTRUCTION C
    close(fd[0]);
    write(1, "message du fils2\n", 17); // INSTRUCTION D
    _exit(0);
  }
  dup2(fd[1],1);
  close(fd[1]); // INSTRUCTION E
  close(fd[0]);
  write(1, "message du pere\n", 16); // INSTRUCTION F
  close(1); // INSTRUCTION G
  while (wait(NULL)>0); // INSTRUCTION H
  return 0;
}
```

Lequel des énoncés suivants est *faux*?

- Le fait d'enlever l'appel système *close* à l'instruction E va normalement mener à un interblocage.
- Le fait d'enlever l'appel système *close* à l'instruction A va normalement mener à un interblocage.
- Le fait d'enlever l'appel système *close* à l'instruction C va normalement mener à un interblocage.
- Le fait d'enlever l'appel système *close* à l'instruction G va normalement mener à un interblocage. ✕

Votre réponse est incorrecte.

La réponse correcte est :
Le fait d'enlever l'appel système *close* à l'instruction C va normalement mener à un interblocage.

Question 10

Correct

Note de 1,00 sur 1,00

Considérez le programme POMME_D'API présenté à la question précédente. Laquelle des opérations suivantes *ne permet pas* de réaliser une lecture ou une écriture au moyen d'un tube anonyme?

- L'appel système *write* à l'instruction B ✓
- L'appel système *write* à l'instruction F
- L'appel système *write* à l'instruction D
- L'appel système *read* à l'instruction R

Votre réponse est correcte.

La réponse correcte est :
L'appel système *write* à l'instruction B

Question 11

Correct

Note de 1,50 sur 1,50

Soit le programme suivant:

```c
#include "fcntl.h"
#include "unistd.h"
#include "errno.h"
#include "stdio.h"

#define BUFSIZE 256

int main()
{
  int fd;
  int nbytes;
  char buf[BUFSIZE];

  fd = open("INF2610_FIFO", O_RDONLY);
  for(;;) {
    if((nbytes = read(fd, buf, BUFSIZE)) <= 0)
      break;
    printf(buf, nbytes);
  }
  return 0;
}
```

Le fichier *INF2610_FIFO* présent dans le répertoire courant correspond à un tube nommé. Vous lancez un processus qui exécute ce programme. Ce processus comporte un seul thread. Tous les appels système fonctionnent correctement.

Lequel des énoncés suivants est *vrai*?

- La fonction *printf* permet d'écrire dans le tampon *buf*, mais il faut savoir que le tampon de *printf* n'est pas nécessairement vidé après chaque écriture.
- Ce processus lecteur pourrait prendre fin "naturellement" (i.e. sans qu'il y ait erreur ou encore réception d'un signal de terminaison) avant que le processus écrivain ferme le tube.
- Le programme de ce processus lecteur est incomplet, car le processus devrait s'assurer de fermer le tube en écriture.
- Il y aura une condition de concurrence entre ce processus lecteur et le processus écrivain, étant donné que le tampon *buf* est une ressource partagée.
- Ce processus lecteur va bloquer sur "open" jusqu'à ce qu'un autre processus ouvre le tube en écriture, à moins que cette ouverture en écriture ne soit déjà faite. ✓

Votre réponse est correcte.

La réponse correcte est :
Ce processus lecteur va bloquer sur "open" jusqu'à ce qu'un autre processus ouvre le tube en écriture, à moins que cette ouverture en écriture ne soit déjà faite.

Question 12

Correct

Note de 1,50 sur 1,50

Complétez le programme suivant que nous appellerons ALARME, en indiquant les instructions aux emplacements /* 1 */, /* 2 */, /* 3 */ et /* 4 */ afin que le message "Alarme déclenchée!" s'afficher au bout de deux secondes.

Note 1: vous pouvez utiliser seulement une seule instruction pour chacun des quatre emplacements, pour un total de quatre instructions.

Note 2: la fonction *alarm* programme une temporisation pour qu'elle envoie un signal SIGALRM au processus appelant au bout du nombre de secondes spécifié. Le traitement par défaut du signal SIGALRM est de terminer le processus destinataire du signal.

Note 3: seule l'expiration de l'alarme doit pouvoir mener à l'affichage du message "Alarme déclenchée!". La réception d'un signal autre que SIGALRM ne doit pas avoir cet effet.

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

/* 1 */

void fonction(int s)
{
  /* 2 */
    /* 3 */
  }
}

int main()
{
  /* 4 */
  printf("On met l'alarme et on attend 2 secondes...\n");
  alarm(2);
  for(;;) if(glob==0) break;
  printf("Alarme déclenchée!\n");
  return 0;
}
```

/* 1 */ [ ] ✓

/* 2 */ [ ] ✓

/* 3 */ [ ] ✓

/* 4 */ [ ] ✓

int s = 2; | int s = 0; | int s = 1; | int glob = 2; | int glob = 1; | int glob = 0;

if(s != glob) { | if(s != SIGALRM) { | if(s == SIGCONT) { | if(glob == 1) { | if(s != SIGCONT) { | if(s == glob) { | if(s == SIGALRM) { | if(glob == 0) {

signal(SIGALRM, fonction); | signal(SIGCONT, glob); | signal(SIGALRM, glob); | _exit(0); | glob = 2 | _exit(1); | glob = 1;

signal(SIGCONT, fonction); | glob = 0;

Votre réponse est correcte.

La réponse correcte est :

Complétez le programme suivant que nous appellerons ALARME, en indiquant les instructions aux emplacements /* 1 */, /* 2 */, /* 3 */ et /* 4 */ afin que le message "Alarme déclenchée!" s'afficher au bout de deux secondes.

Note 1: vous pouvez utiliser seulement une seule instruction pour chacun des quatre emplacements, pour un total de quatre instructions.

Note 2: la fonction *alarm* programme une temporisation pour qu'elle envoie un signal SIGALRM au processus appelant au bout du nombre de secondes spécifié. Le traitement par défaut du signal SIGALRM est de terminer le processus destinataire du signal.

Note 3: seule l'expiration de l'alarme doit pouvoir mener à l'affichage du message "Alarme déclenchée!". La réception d'un signal autre que SIGALRM ne doit pas avoir cet effet.

```c
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

/* 1 */

void fonction(int s)
{
  /* 2 */
    /* 3 */
  }
}

int main()
{
  /* 4 */
  printf("On met l'alarme et on attend 2 secondes...\n");
  alarm(2);
  for(;;) if(glob==0) break;
  printf("Alarme déclenchée!\n");
  return 0;
}
```

/* 1 */ [int glob = 1;]

/* 2 */ [if(s == SIGALRM) {]

/* 3 */ [glob = 0;]

/* 4 */ [signal(SIGALRM, fonction);]

Question 13

Correct

Note de 1,50 sur 1,50

Considérez les processus A, B et C suivants :

Sémaphore x = 0, y=1;

A	B	C
P(y);	P(x);	P(y);
a;	b1;	P(x);
V(x);	b2;	c;
V(y);	V(x);	V(x);

Sélectionnez un ordre possible d'exécution des actions atomiques a, b1, b2, et c, sachant que ces trois processus s'exécutent en concurrence.

- c; b1; b2; a;
- a; b1; b2; c; ✓
- a; b1; c; b2;
- c; a; b1; b2;
- b1; b2; a; c;

Votre réponse est correcte.

La réponse correcte est :
a; b1; b2; c;

Question 14

Correct

Note de 1,50 sur 1,50

Complétez la fonction suivante en indiquant les instructions aux emplacements /* 1 */, /* 2 */ et /* 3 */, afin que l'incrémentation du compteur se fasse en évitant les conditions de concurrence. Note: vous pouvez utiliser seulement une seule instruction à l'emplacement /* 1 */, une seule instruction à l'emplacement /* 2 */, et une seule instruction à l'emplacement /* 3 */. Une seule solution est possible.

```
#include <pthread.h>
#include <stdio.h>
#include <stdatomic.h>

/* 1 */
int un = 1, moinsun = -1;

void * inc_dec(void * x)
{
  int i, pas = *(int *) x;
  for(i = 0; i < 10000000; i++) {
    /* 2 */
  }

  printf("fonction inc_dec(%d), compteur = %d\n", pas, compteur);
}

int main()
{
  /* 3 */
  pthread_t t1, t2;
  pthread_create(&t1, NULL, inc_dec, &un);
  pthread_create(&t2, NULL, inc_dec, &moinsun);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  return 0;
}
```

/* 1 */ [ ] ✓

/* 2 */ [ ] ✓

/* 3 */ [ ] ✓

int compteur; | volatile atomic_long compteur; | pthread_spinlock_t lock; | sem_t s; | int lock;

sem_wait(&s); | pthread_spin_unlock(&lock); | pthread_spin_lock(&lock); | while(TSL(lock)!=0); | lock = 1; | sem_post(&s); | lock = 0;

atomic_fetch_add(&compteur, pas);

compteur = 0; | s = 0; | atomic_init(&compteur, 0); | lock = 0; | lock = 1; | s = 1;

Votre réponse est correcte.

La réponse correcte est :

Complétez la fonction suivante en indiquant les instructions aux emplacements /* 1 */, /* 2 */ et /* 3 */, afin que l'incrémentation du compteur se fasse en évitant les conditions de concurrence. Note: vous pouvez utiliser seulement une seule instruction à l'emplacement /* 1 */, une seule instruction à l'emplacement /* 2 */, et une seule instruction à l'emplacement /* 3 */. Une seule solution est possible.

```c
#include <pthread.h>
#include <stdio.h>
#include <stdatomic.h>

/* 1 */
int un = 1, moinsun = -1;

void * inc_dec(void * x)
{
  int i, pas = *(int *) x;
  for(i = 0; i < 10000000; i++) {
    /* 2 */
  }

  printf("fonction inc_dec(%d), compteur = %d\n", pas, compteur);
}

int main()
{
  /* 3 */
  pthread_t t1, t2;
  pthread_create(&t1, NULL, inc_dec, &un);
  pthread_create(&t2, NULL, inc_dec, &moinsun);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  return 0;
}
```

/* 1 */ [volatile atomic_long compteur;]
/* 2 */ [atomic_fetch_add(&compteur, pas);]
/* 3 */ [atomic_init(&compteur, 0);]

Question 15

Correct

Note de 1,50 sur 1,50

Nous vous présentons quatre extraits d'algorithmes pour le problème des philosophes. Lequel de ces extraits est le plus susceptible d'éviter à la fois les interblocages et les famines?

Note 1: tous ces extraits sont exempts d'erreur, mais seuls trois d'entre eux préviennent adéquatement les interblocages et les famines.

Note 2: pour cette question, ne tenez pas compte du fait que les philosophes puissent manger en même temps ou pas; i.e. il est acceptable que les threads ne s'exécutent pas en concurrence.

Note 3: les déclarations globales suivantes sont faites pour tous les extraits présentés:

```
#define N 5
#define G (i+1)%N
#define D i
#define libre 1
#define occupe 0

int fourchettes[N] = {libre, libre, libre, libre, libre};
Semaphore f[N] = {1, 1, 1, 1, 1};
Semaphore s[N] = {1, 0, 0, 0, 0};
sem_t mutex;
```

- ✓
```
void philosophe(int i)
{
  sleep(1); // penser
  sem_wait(&mutex);
  if(fourchettes[G] == libre && fourchettes[D] == libre) {
    fourchettes[G] = occupe;
    fourchettes[D] = occupe;
    sem_post(&mutex);
    print("philosophe %d mange\n", i);
    sleep(1); // manger
    print("philosophe %d a fini\n", i);
  //...
}
```

-
```
void philosophe(int i)
{
  sleep(1); // penser
  sem_wait(s[i]);
  print("philosophe %d mange\n", i);
  sleep(1); // manger
  print("philosophe %d a fini\n", i);
  sem_post(s[(i+1)%N]);
  //...
}
```

-
```
void philosophe(int i)
{
  sleep(1); // penser
  if(G<D) { P(f[G]); P(f[D]); }
  else { P(f[D]); P(f[G]); }
  print("philosophe %d mange\n", i);
  sleep(1); // manger
  print("philosophe %d a fini\n", i);
  //...
}
```

```
void philosophe(int i)
{
  sleep(1); // penser
  if(G>D) { P(f[G]); P(f[D]); }
  else { P(f[D]); P(f[G]); }
  print("philosophe %d mange\n", i);
  sleep(1); // manger
  print("philosophe %d a fini\n", i);
  //...
}
```

Votre réponse est correcte.

La réponse correcte est :

```
void philosophe(int i)
{
  sleep(1); // penser
  sem_wait(&mutex);
  if(fourchettes[G] == libre && fourchettes[D] == libre) {
    fourchettes[G] = occupe;
    fourchettes[D] = occupe;
    sem_post(&mutex);
    print("philosophe %d mange\n", i);
    sleep(1); // manger
    print("philosophe %d a fini\n", i);
  //...
}
```

Question 16

Incorrect

Note de 0,00 sur 1,50

Nous vous présentons les quatre extraits d'algorithmes pour le problème des philosophes présentés à la question précédente. Un seul de ces extraits *ne permet pas* aux threads de s'exécuter en concurrence. Lequel?

Note: les déclarations globales suivantes (les mêmes que pour la dernière question) sont faites pour tous les extraits présentés:

```
#define N 5
#define G (i+1)%N
#define D i
#define libre 1
#define occupe 0

int fourchettes[N] = {libre, libre, libre, libre, libre};
Semaphore f[N] = {1, 1, 1, 1, 1};
Semaphore s[N] = {1, 0, 0, 0, 0};
sem_t mutex;
```

◉ ✗
```
void philosophe(int i)
{
  sleep(1); // penser
  if(G<D) { P(f[G]); P(f[D]); }
  else { P(f[D]); P(f[G]); }
  print("philosophe %d mange\n", i);
  sleep(1); // manger
  print("philosophe %d a fini\n", i);
  //...
}
```

○
```
void philosophe(int i)
{
  sleep(1); // penser
  sem_wait(s[i]);
  print("philosophe %d mange\n", i);
  sleep(1); // manger
  print("philosophe %d a fini\n", i);
  sem_post(s[(i+1)%N]);
  //...
}
```

○
```
void philosophe(int i)
{
  sleep(1); // penser
  if(G>D) { P(f[G]); P(f[D]); }
  else { P(f[D]); P(f[G]); }
  print("philosophe %d mange\n", i);
  sleep(1); // manger
  print("philosophe %d a fini\n", i);
  //...
}
```

```
void philosophe(int i)
{
  sleep(1); // penser
  sem_wait(&mutex);
  if(fourchettes[G] == libre && fourchettes[D] == libre) {
    fourchettes[G] = occupe;
    fourchettes[D] = occupe;
    sem_post(&mutex);
```

Votre réponse est incorrecte.

La réponse correcte est :

```
void philosophe(int i)
{
  sleep(1); // penser
  sem_wait(s[i]);
  print("philosophe %d mange\n", i);
  sleep(1); // manger
  print("philosophe %d a fini\n", i);
  sem_post(s[(i+1)%N]);
  //...
}
```