Statut	Terminé
Commencé	lundi 28 octobre 2024, 18:00
Terminé	lundi 28 octobre 2024, 20:00
Durée	2 heures
Note	**29,00** sur 30,00 (**96,67**%)

Question 1

Non répondue

Non noté

Directives :

1. Cet examen est composé de 15 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. Certains étudiants en échange dont la maîtrise du français est imparfaite pourront amener un dictionnaire en format papier seulement. Celui-ci devra être exempt d'annotations. Veuillez contacter le professeur à l'avance si vous souhaitez bénéficier de cet accommodement.
5. Aucune réponse aux questions durant l'examen.
6. **Sauf mention contraire, tous les appels système utilisés dans les questions sont supposés exempts d'erreurs et considèrent leurs options par défaut**.
7. Il n'est pas demandé de traiter les cas d'erreurs, ni d'inclure les directives d'inclusion dans les codes à compléter.
8. Pour les questions à choix multiples, **vous devez sélectionner une seule réponse**.
9. Il n'est pas possible de joindre des fichiers à vos réponses.
10. Lisez au complet chaque question avant d'y répondre.
11. Vous pouvez inclure vos commentaires au responsable du cours dans l'espace ci-après.

Bon examen à tous

Question 3

Correct

Note de 2,00 sur 2,00

Lequel de ces énoncés est *faux*?

- Le va-et-vient (ou "swapping") consiste à retirer de la mémoire principale les travaux bloqués pour les remplacer par des travaux à placer immédiatement en état "exécution". ✓
- L'invention des contrôleurs DMA et des unités de disque a rendu possible l'avènement de la multiprogrammation.
- La principale différence entre le traitement par lot et le partage de temps est que, en partage de temps, le système d'exploitation suspend l'exécution du travail en cours si ce dernier a consommé son temps d'allocation du processeur.
- Avant l'avènement de la multiprogrammation, les ordinateurs subissaient des pertes de temps importantes du fait qu'un seul travail était présent en mémoire à la fois.
- Le parallélisme, contrairement au pseudo-parallélisme, nécessite la présence d'autant de processeurs que de tâches à exécuter en parallèle.

Votre réponse est correcte.

La réponse correcte est :
Le va-et-vient (ou "swapping") consiste à retirer de la mémoire principale les travaux bloqués pour les remplacer par des travaux à placer immédiatement en état "exécution".

Question 4

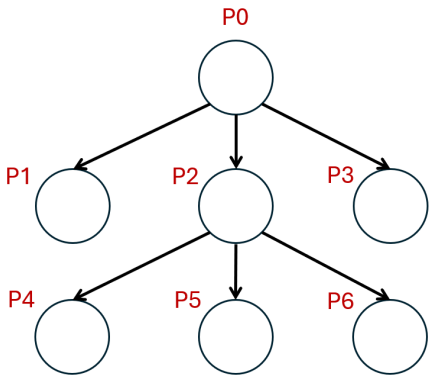
Correct

Note de 3,00 sur 3,00

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 P6). **Rappel : les appels système fonctionnent et ne retournent aucune erreur.**

Assurez-vous de prendre connaissance des instructions ci-après qui sont telles que définies; vous n'avez pas à les insérer dans votre solution.

int main() {
/*0*/ **char id_processus[7][3] = {"P0", "P1", "P2", "P3", "P4", "P5", "P6"};** <-- *L'instruction /*0*/ est nécessairement telle que définie ici.*
/*1*/
/*2*/
/*3*/
/*4*/
/*5*/
/*6*/
/*7*/ **}** <-- *L'instruction /*7*/ est nécessairement une accolade fermante.*
/*8*/ **}** <-- *L'instruction /*8*/ est nécessairement une accolade fermante.*
/*9*/
/*10*/
/*11*/
/*12*/
/*13*/ **}** <-- *L'instruction /*13*/ est nécessairement une accolade fermante.*
/*14*/ **}** <-- *L'instruction /*14*/ est nécessairement une accolade fermante.*
/*15*/ **}** <-- *L'instruction /*15*/ est nécessairement une accolade fermante.*
/*16*/
/*17*/ **_exit(0);** <-- *L'instruction /*17*/ est nécessairement _exit(0);.*
} <-- accolade fermante du main.



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

/*1*/	for(int=1; i<=3; i++) {	✓
/*2*/	if(fork()==0) {	✓
/*3*/	if(i==2) {	✓
/*4*/	for(int j=4; j<=6; j++) {	✓
/*5*/	if(fork()==0) {	✓

/*6*/	execlp("echo", "echo", id_processus[j], NULL);	✓
/*9*/	while(wait(NULL)>0);	✓
/*10*/	_exit(0);	✓
/*11*/	} else {	✓
/*12*/	execlp("echo", "echo", id_processus[i], NULL);	✓
/*16*/	while(wait(NULL)>0);	✓

Votre réponse est correcte.

La réponse correcte est :
/*1*/ → for(int=1; i<=3; i++) {,

/*2*/ → if(fork()==0) {,

/*3*/ → if(i==2) {,

/*4*/ → for(int j=4; j<=6; j++) {,

/*5*/ → if(fork()==0) {,

/*6*/ → execlp("echo", "echo", id_processus[j], NULL);,

/*9*/ → while(wait(NULL)>0);,

/*10*/ → _exit(0);,

/*11*/ → } else {,

/*12*/ → execlp("echo", "echo", id_processus[i], NULL);,

/*16*/ → while(wait(NULL)>0);

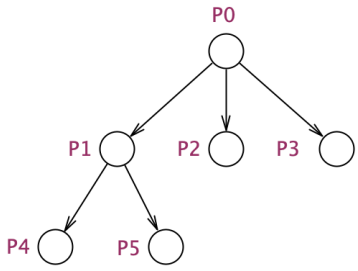
Question 5

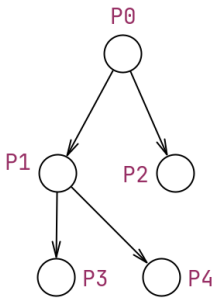
Correct

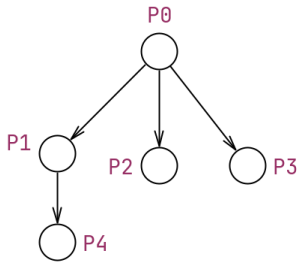
Note de 2,00 sur 2,00

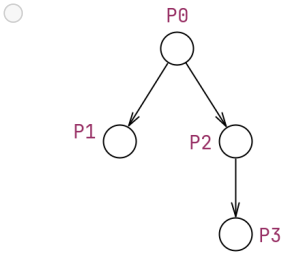
Quel arbre représente la hiérarchie de processus générés par la fonction suivante (où P0 est le processus appelant de la fonction construireArbre) :

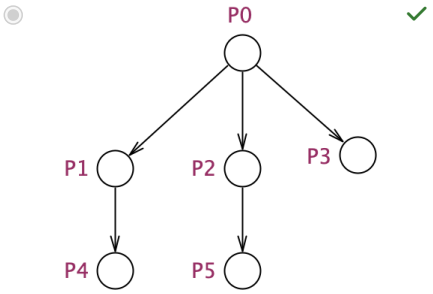
```
void construireArbre()
{
  for(int i=0; i<3; i++) {
    if(fork()==0) {
      if(i == 2) break;
      fork();
      break;
    }
  }
  while(wait(NULL)>0);
  _exit(0);
}
```

- Aucune de ces réponses.
- 

- 

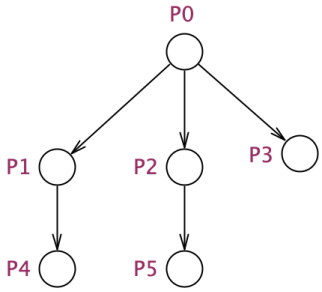
- 





Votre réponse est correcte.

La réponse correcte est :



Question 6

Correct

Note de 3,00 sur 3,00

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 exécutez le programme en fournissant un entier comme argument. La sortie suivante s'affiche:

```
i=1; ID processus = 8399; ID parent = 8085; ID enfant = 8400
i=2; ID processus = 8400; ID parent = 7360; ID enfant = 8401
i=3; ID processus = 8401; ID parent = 8400; ID enfant = 8402
i=4; ID processus = 8402; ID parent = 7360; ID enfant = 8403
i=5; ID processus = 8403; ID parent = 8402; ID enfant = 8405
i=6; ID processus = 8405; ID parent = 7360; ID enfant = 0
```

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

- Sur cette installation, les processus orphelins sont adoptés par le processus *init*.
- Le processus portant le PID 8399 s'est terminé avant que le processus portant le PID 8400 complète son exécution. ✓
- La valeur de l'entier fourni en argument est de 5.
- Le processus *systemd* a un PID de 8085.
- Le processus *bash* a un PID de 7360

Votre réponse est correcte.

La réponse correcte est :
Le processus portant le PID 8399 s'est terminé avant que le processus portant le PID 8400 complète son exécution.

Question 7

Correct

Note de 2,00 sur 2,00

Lequel des énoncés suivants est *faux*?

- En mode d'annulation retardé, 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 les compilateurs et bibliothèques d'exécution.
- Lorsqu'un thread appelle *fork()*, tous les threads du processus sont copiés, pas seulement le thread appelant. ✓
- 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.
- Les threads de la bibliothèque *pthreads* sont préemptifs.

Votre réponse est correcte.

La réponse correcte est :
Lorsqu'un thread appelle *fork()*, tous les threads du processus sont copiés, pas seulement le thread appelant.

Question 8

Correct

Note de 2,00 sur 2,00

Le thread principal d'un processus crée 5 threads, puis se met en attente de la fin d'exécution de ceux-ci. Lequel des énoncés suivants est *faux*?

- S'il s'agit de threads utilisateurs, les variables globales ne seront pas sujettes aux conditions de concurrence entre les threads.
- Typiquement, le temps de création de chaque thread sera plus élevé s'il s'agit de threads noyau, et moins élevé s'il s'agit de threads utilisateur.
- Les threads peuvent communiquer entre eux sans passer par des appels système et ce, peu importe qu'il s'agisse de threads utilisateur ou de threads noyau.
- L'accès à une variable locale créée par l'un des threads sera partagé par tous les threads. ✓
- S'il s'agit de threads noyau, il pourrait y avoir plus d'un thread du processus en cours d'exécution sur des processeurs différents.

Votre réponse est correcte.

La réponse correcte est : L'accès à une variable locale créée par l'un des threads sera partagé par tous les threads.

Question 9

Correct

Note de 2,00 sur 2,00

Considérez le programme suivant que nous appellerons ÉCLAIR, dont la mission est de produire la chaine "ABCDEFGHIJKLMNOPQRSTUVWXYZ\n\n". La compilation de ce programme ne génère pas d'erreur, mais la sortie du programme semble problématique. (Des numéros de ligne ont été ajoutés mais ne font pas partie du programme.)

```
  int main () {
01 char msg1[] = "ACEGIKMOQSUWY\n";
02 char msg2[] = "BDFHJLNPRTVXZ\n";
03
04 sem_t * S1 = sem_open("/semaphore1", O_CREAT, 0600, 0);
05 sem_t * S2 = sem_open("/semaphore2", O_CREAT, 0600, 0);
06
07
08
09 int fd[2]; pipe(fd);
10 if(fork()==0)  {
11   dup2(fd[1],1);
12   close(fd[1]);
13   close(fd[0]);
14   for(int i=0; i<strlen(msg1); i++) {
15
16     write(1, msg1+i, 1);
17
18   }
19   _exit(0);
20 }
21 if(fork()==0)  {
22   dup2(fd[1],1);
23   close(fd[1]);
24   close(fd[0]);
25   for(int i=0; i<strlen(msg2); i++) {
26
27     write(1, msg2+i, 1);
28
29   }
30   _exit(0);
31 }
32 dup2(fd[0],0);
33 close(fd[1]);
34 close(fd[0]);
35 char c;
36 while (read(0,&c,1) >0)
37   write(1,&c, 1);
38 close(1);
39 wait(NULL);
40 wait(NULL);
41 return 0;
 }
```

Quelles sorties sont-elles possibles parmi les suivantes?

- Le programme n'affichera rien.
- Deux sorties possibles seulement: AAAAAAAAAAAAAABBBBBBBBBBBBBB et BBBBBBBBBBBBBBAAAAAAAAAAAAAA.

- Deux sorties possibles seulement: ACEGIKMOQSUWY\nBDFHJLNPRTVXZ\n et BDFHJLNPRTVXZ\nACEGIKMOQSUWY\n.
- Plusieurs sorties sont possibles, qu'il serait trop long d'énumérer. ✓
- Deux sorties possibles seulement: ABCDEFGHIJKLMNOPQRSTUVWXYZ\n\n et BADCFEHGJILKNMPORQTSVUXWZY\n\n.

Votre réponse est correcte.

La réponse correcte est : Plusieurs sorties sont possibles, qu'il serait trop long d'énumérer.

Question 10

Correct

Note de 2,00 sur 2,00

Considérez le programme ÉCLAIR vu à la question précédente. Complétez le programme de manière à ce qu'il affiche à l'écran la chaine "ABCDEFGHIJKLMNOPQRSTUVWXYZ\n\n" tel que prévu. IMPORTANT: assurez-vous que le premier enfant bloque sur S1 et non pas sur S2.

```
int main () {
  char msg1[] = "ACEGIKMOQSUWY\n";
  char msg2[] = "BDFHJLNPRTVXZ\n";

  sem_t * S1 = sem_open("/semaphore1", O_CREAT, 0600, 0);
  sem_t * S2 = sem_open("/semaphore2", O_CREAT, 0600, 0);
  sem_init(S1, 1, 1);
  sem_init(S2, 1, 0);
  int fd[2]; pipe(fd);
  if(fork()==0)  {
    dup2(fd[1],1);
    close(fd[1]);
    close(fd[0]);
    for(int i=0; i<strlen(msg1); i++) {
      sem_wait(S1);
      write(1, msg1+i, 1);
      sem_post(S2);
    }
    _exit(0);
  }
  if(fork()==0)  {
    dup2(fd[1],1);
    close(fd[1]);
    close(fd[0]);
    for(int i=0; i<strlen(msg2); i++) {
      sem_wait(S2);
      write(1, msg2+i, 1);
      sem_post(S1);
    }
    _exit(0);
  }
  dup2(fd[0],0);
  close(fd[1]);
  close(fd[0]);
  char c;
  while (read(0,&c,1) >0)
    write(1,&c, 1);
  close(1);
  wait(NULL);
  wait(NULL);
  return 0;
}
```

sem_init(S1, 1, 0);

sem_init(S2, 1, 1);

sem_trywait(S1);

sem_wait(S2);

sem_trywait(S2);

sem_wait(S1);

Votre réponse est correcte.

Question 11

Correct

Note de 1,00 sur 1,00

Dans le programme ÉCLAIR que vous venez de modifier à la question précédente, est-il possible de remplacer l'utilisation des sémaphores par celle des verrous actifs, tout en conservant l'allure générale de la solution? Sélectionnez la *meilleure* réponse.

- Non, car cela causera certainement un interblocage.
- Non, car cela risque de causer un interblocage.
- Oui, ce qui permettra d'éviter les inversions de priorité.
- Oui, mais avec une probable augmentation du temps d'utilisation du processeur.
- Non, car les verrous actifs servent plutôt à assurer l'exclusion mutuelle. ✓
- Oui, ce qui permettra d'éviter un certain nombre de changements de contexte.

Votre réponse est correcte.

La réponse correcte est :
Non, car les verrous actifs servent plutôt à assurer l'exclusion mutuelle.

Question 12

Incorrect

Note de 0,00 sur 1,00

Toujours dans le contexte du programme ÉCLAIR vu précédemment *et tel que complété*, lequel des énoncés suivants est *faux*?

- L'utilisation de tubes nommés à la place de tubes anonymes serait nécessaire dans le cas où l'on souhaiterait utiliser des processus non-apparentés.
- L'utilisation de tubes nommés à la place de tubes anonymes nécessiterait de porter attention à l'ordre d'ouverture des tubes et ce, afin d'éviter les interblocages.
- L'ouverture des sémaphores avec sem_open est rendue nécessaire par le fait qu'ils sont utilisés à travers des processus différents.
- Le fait de déplacer le premier *wait* tout de suite avant le deuxième *fork()* causerait forcément un interblocage. ✗
- Si on utilise des tubes nommés à la place des tubes anonymes, chacun d'eux sera détruit après la fermeture de tous ses descripteurs de fichiers.

Votre réponse est incorrecte.

La réponse correcte est :
Si on utilise des tubes nommés à la place des tubes anonymes, chacun d'eux sera détruit après la fermeture de tous ses descripteurs de fichiers.

Question 13

Correct

Note de 2,00 sur 2,00

Lequel des énoncés suivants est *vrai*?

- Le signal SIGINT ne peut être masqué.
- La commande *signal (SIGINT, SIG_IGN)* assigne le gestionnaire de signal par défaut au processus courant pour le signal SIGINT.
- Si un gestionnaire de signal sur mesure est assigné à un signal pour un processus donné, la réception de ce signal par ce processus va provoquer l'exécution de ce gestionnaire, puis à l'arrêt du processus.
- Le signal SIGCHLD ne peut être masqué.
- La commande *pause()* met le processus courant en suspens jusqu'à la réception d'un signal, ce qui mènera ensuite à l'exécution du comportement défini pour ce signal. ✓

Votre réponse est correcte.

La réponse correcte est :
La commande *pause()* met le processus courant en suspens jusqu'à la réception d'un signal, ce qui mènera ensuite à l'exécution du comportement défini pour ce signal.

Question 14

Correct

Note de 2,00 sur 2,00

Considérez les processus A, B et C suivants :

Sémaphore x = 1, y=1;

A	B	C
P(x);	P(y);	P(x);
P(y);	b1;	c1;
a1;	V(x);	V(x);
a2;	b2;	P(y);
V(x);	P(x);	c2;
V(y);	V(y)	V(y);

Sélectionnez un ordre possible d'exécution des actions atomiques a1, a2, b1, b2, c1 et c2, sachant que ces trois processus s'exécutent en concurrence.

- a1; b1; a2; b2; c1; c2;
- c1; a1; a2; b1; c2; b2;
- b1; c1; b2; a1; a2; c2; ✓
- a1; a2; b1; c1; c2; b2;
- b1; b2; c1; a1; c2; a2;

Votre réponse est correcte.

La réponse correcte est :
b1; c1; b2; a1; a2; c2;

Question 15

Correct

Note de 2,00 sur 2,00

Complétez la fonction suivante, appelée INCRÉMENT, afin que l'incrémentation du compteur se fasse en évitant les conditions de concurrence. Veuillez utiliser comme seul mécanisme de synchronisation le *verrou actif*. La valeur initiale du compteur sera égale à zéro. Une seule solution est possible.

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

int decompte = 0;
pthread_spinlock_t verrou;
int positif = 1, negatif = -1;

void * ajout(void * x)
{
  int i, delta1 = *(int *) x;
  for(i = 0; i < 10000000; i++) {
    pthread_spin_lock(&verrou);
    decompte = decompte + delta1;
    pthread_spin_unlock(&verrou);
  }
  printf("fonction inc_dec(%d), compteur = %d\n", delta1, decompte);
  pthread_exit(NULL);
}

int main()
{
  pthread_spin_init(&verrou, PTHREAD_PROCESS_PRIVATE);
  pthread_t t1, t2;
  pthread_create(&t1, NULL, ajout, &positif);
  pthread_create(&t2, NULL, ajout, &negatif);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  return 0;
}
```

volatile atomic_long decompte;

sem_t verrou;

pthread_mutex_t verrou=PTHREAD_MUTEX_INITIALIZER

atomic_init(&decompte, 0);

verrou = 1;

atomic_fetch_add(&decompte, delta1);

pthread_mutex_lock(&verrou);

while(TSL(verrou)!=0);

sem_post(&verrou);

sem_wait(&verrou);

pthread_mutex_unlock(&verrou);

verrou = 0;

Votre réponse est correcte.

Question 16

Correct

Note de 2,00 sur 2,00

Considérez la fonction INCRÉMENT vue à la question précédente. Laquelle des sorties suivantes est la seule sortie possible?

- fonction inc_dec(-1), compteur = -9971561 ✓
  fonction inc_dec(1), compteur = 0
- fonction inc_dec(1), compteur = 9162229
  fonction inc_dec(-1), compteur = 4271
- fonction inc_dec(-1), compteur = 0
  fonction inc_dec(1), compteur = 7154
- fonction inc_dec(1), compteur = -9920856
  fonction inc_dec(-1), compteur = 0
- fonction inc_dec(1), compteur = 0
  fonction inc_dec(-1), compteur = 8697340

Votre réponse est correcte.

La réponse correcte est :
fonction inc_dec(-1), compteur = -9971561
fonction inc_dec(1), compteur = 0

Question 17

Correct

Note de 2,00 sur 2,00

Cette question porte sur la 2^{e} solution au problème des lecteurs-rédacteurs. Lequel des énoncés suivants est *vrai*?

- Le rédacteur sera mis en attente si le tampon et plein, et le lecteur sera mis en attente si le tampon est vide.
- Cette solution comporte un risque de famine pour les lecteurs, mais pas de risque de famine pour les écrivains.
- Cette solution permet la présence d'un seul lecteur à la fois, étant donné que P(redact) et V(redact) encadrent la fonction lire().
- Cette solution est sujette aux interblocages.
- Le sémaphore *tour* vise à empêcher la présence simultanée de rédacteurs et de lecteur.
- Les manipulations de *nbL* doivent se faire en sections critiques. On protège celles-ci par *mutex*. ✓

Votre réponse est correcte.

La réponse correcte est :
Les manipulations de *nbL* doivent se faire en sections critiques. On protège celles-ci par *mutex*.