Programmation Concurrente

Principes et concepts
2024-2025

Chapitre3: Synchronisation

Plan

- Rappel
- Scenario d'alternance
- Déterminisme
- Atomicité
- Exclusion mutuelle

Ordre des opérations

- Soit les threads A et B suivants s'exécutant concurremment et les variables globales `a` et `b`:

A

```
a = a + 1
b = b + 1
```

B

```
b = 2 * b
a = 2 * a
```

- Si `a = b = 2`, quelles seront les valeurs de `a` et `b` lorsque les deux threads seront terminés ?

Ordre des opérations

- Soit les threads A et B suivants s'exécutant concurremment et les variables globales `a` et `b`:

A

```
a = a + 1
b = b + 1
```

B

```
b = 2 * b
a = 2 * a
```

1

```
a = a + 1
b = b + 1
b = 2 * b
a = 2 * a

a = 6, b = 6
```

Ordre des opérations

- Soit les threads A et B suivants s'exécutant concurremment et les variables globales `a` et `b`:

A

```
a = a + 1
b = b + 1
```

B

```
b = 2 * b
a = 2 * a
```

1

```
a = a + 1
b = b + 1
b = 2 * b
a = 2 * a

a = 6, b = 6
```

2

```
b = 2 * b
a = 2 * a
a = a + 1
b = b + 1

a = 5, b = 5
```

Ordre des opérations

- Soit les threads A et B suivants s'exécutant concurremment et les variables globales `a` et `b`:

A

```
a = a + 1
b = b + 1
```

B

```
b = 2 * b
a = 2 * a
```

1

```
a = a + 1
b = b + 1
b = 2 * b
a = 2 * a

a = 6, b = 6
```

2

```
b = 2 * b
a = 2 * a
a = a + 1
b = b + 1

a = 5, b = 5
```

3

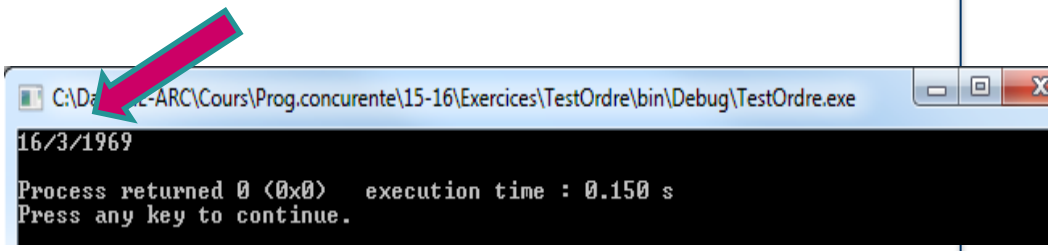
```
a = a + 1
b = 2 * b
b = b + 1
a = 2 * a

a = 6, b = 5
```

Ordre des opérations

```
int day, month, year; // date

void *func1(void *arg) {
        day = 28;
        month = 12;
        year = 1969;
        return NULL;
}

void *func2(void *arg) {
        day = 16;
        month = 3;
        year = 1953;
        return NULL;
}

int main(int argc, char *argv[]) {
        pthread_t thread1, thread2;
        pthread_create(&thread1, NULL, func1, NULL);
        pthread_create(&thread2, NULL, func2, NULL);
        pthread_join(thread1, NULL);
        pthread_join(thread2, NULL);
        printf("%d/%d/%d\n", day, month, year);
        return EXIT_SUCCESS;
}
```

Exemple1

```
int day, month, year; // date

void *func1(void *arg) {
        day = 28;
        month = 12;
        year = 1969;
        return NULL;
}

void *func2(void *arg) {
        day = 16;
        month = 3;
        year = 1953;
        return NULL;
}

int main(int argc, char *argv[]) {
        pthread_t thread1, thread2;
        pthread_create(&thread1, NULL, func1, NULL);
        pthread_create(&thread2, NULL, func2, NULL);
        pthread_join(thread1, NULL);
        pthread_join(thread2, NULL);
        printf("%d/%d/%d\n", day, month, year);
        return EXIT_SUCCESS;
}
```

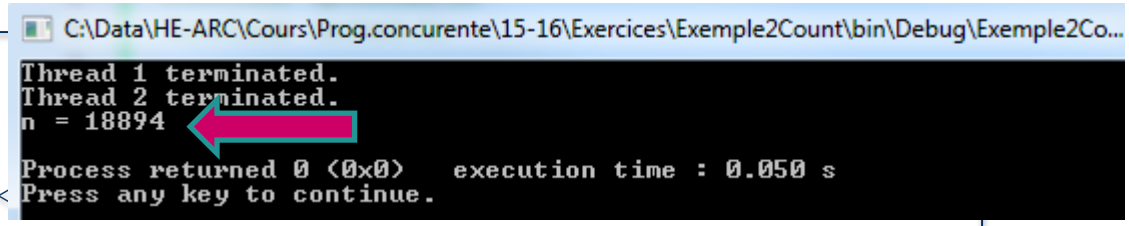


Exemple2

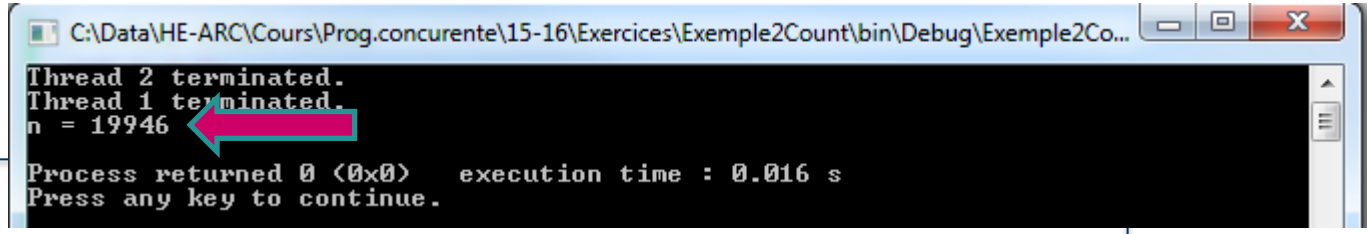
```
#define COUNT 10000
int n = 0;

void *func1(void *arg) {
        for (int i = 0; i < COUNT; i++)
                n++;
        printf("Thread 1 terminated.\n");
        return NULL;
}

void *func2(void *arg) {
        for (int i = 0; i < COUNT; i++)
                n++;
        printf("Thread 2 terminated.\n");
        return NULL;
}

int main(int argc, char *argv[]) {
        pthread_t thread1, thread2;
        pthread_create(&thread1, NULL, func1, NULL);
        pthread_create(&thread2, NULL, func2, NULL);
        pthread_join(thread1, NULL);
        pthread_join(thread2, NULL);
        printf("n = %d\n", n);
        return EXIT_SUCCESS;
}
```

Exemple2

```
#define COUNT 10000
int n = 0;

void *func1(void *arg) {
        for (int i = 0; i <
                n++;
        printf("Thread 1 terminated.\n");
        return NULL;
}

void *func2(void *arg) {
        for (int i = 0; i < COUNT; i++)
                n++;
        printf("Thread 2 terminated.\n");
        return NULL;
}

int main(int argc, char *argv[]) {
        pthread_t thread1, thread2;
        pthread_create(&thread1, NULL, func1, NULL);
        pthread_create(&thread2, NULL, func2, NULL);
        pthread_join(thread1, NULL);
        pthread_join(thread2, NULL);
        printf("n = %d\n", n);
        return EXIT_SUCCESS;
}
```



Exemple2




```
#define COUNT 10000
int n = 0;

void *func1(void *arg) {
        for (int i = 0; i < COUNT; i++)
                n++;
        printf("Thread 1 terminated.\n");
        return NULL;
}

void *func2(void *arg) {
        for (int i = 0; i < COUNT; i++)
                n++;
        printf("Thread 2 terminated.\n");
        return NULL;
}

int main(int argc, char *argv[]) {
        pthread_t thread1, thread2;
        pthread_create(&thread1, NULL, func1, NULL);
        pthread_create(&thread2, NULL, func2, NULL);
        pthread_join(thread1, NULL);
        pthread_join(thread2, NULL);
        printf("n = %d\n", n);
        return EXIT_SUCCESS;
}
```

Explication

- On pense obtenir 20000, mais ce n'est le cas; pourquoi ?
- Que signifie n++ ?
- Cette simple instruction C d'incrémentation est réellement composée de 3 instructions en langage machine :

Concurrence et déterminisme

- Soit n = 10
- L'instruction n++ peut être interrompue :

Thread 1	Thread 2	
`eax ← 10`		

Concurrence et déterminisme

- Soit n = 10
- L'instruction n++ peut être interrompue :

Thread 1	Thread 2	
`eax ← 10`		
		Changement de contexte

temps

Concurrence et déterminisme

- Soit n = 10
- L'instruction n++ peut être interrompue :

Thread 1	Thread 2	
`eax ← 10`		
		Changement de contexte
	`eax ← 10`	

temps

Concurrence et déterminisme

- Soit n = 10
- L'instruction n++ peut être interrompue :

Thread 1	Thread 2	
`eax ← 10`		
		Changement de contexte
	`eax ← 10`	
	`eax ← eax + 1`	

temps

Concurrence et déterminisme

- Soit n = 10
- L'instruction n++ peut être interrompue :

Thread 1	Thread 2	
`eax ← 10`		
		Changement de contexte
	`eax ← 10`	
	`eax ← eax + 1`	
	`n ← eax` (vaut 11)	

temps

Concurrence et déterminisme

- Soit n = 10
- L'instruction n++ peut être interrompue :

Thread 1	Thread 2	
`eax ← 10`		
		Changement de contexte
	`eax ← 10`	
	`eax ← eax + 1`	
	`n ← eax` (vaut 11)	
		Changement de contexte

temps

Concurrence et déterminisme

- Soit n = 10
- L'instruction n++ peut être interrompue :

Thread 1	Thread 2	
`eax ← 10`		
		Changement de contexte
	`eax ← 10`	
	`eax ← eax + 1`	
	`n ← eax` (vaut 11)	
		Changement de contexte
`eax ← eax + 1`		

temps

Concurrence et déterminisme

- Soit n = 10
- L'instruction n++ peut être interrompue :

Thread 1	Thread 2	
`eax ← 10`		
		Changement de contexte
	`eax ← 10`	
	`eax ← eax + 1`	
	`n ← eax` (vaut 11)	
		Changement de contexte
`eax ← eax + 1`		
`n ← eax`		

temps

Concurrence et déterminisme

- Soit n = 10
- L'instruction n++ peut être interrompue :

Thread 1	Thread 2	
`eax ← 10`		
		Changement de contexte
	`eax ← 10`	
	`eax ← eax + 1`	
	`n ← eax` (vaut 11)	
		Changement de contexte
`eax ← eax + 1`		
`n ← eax`		
		`n` vaut 11 et non 12 !

temps

Atomicité

- L'exemple d'exécution précédent montre que la variable `n` ne peut pas être utilisée comme compteur.
- Pour résoudre ce problème l'instruction `n++` doit s'exécuter de manière **atomique**, c'est-à-dire que les deux accès à la variable `n` doivent s'effectuer de manière **indivisible**.
- Une portion de code qui doit s'exécuter de manière atomique est appelée une **section critique** (SC).
- Une solution consiste à inclure et protéger la section critique à l'aide d'un verrou (ou mutex).

Atomicité

Une **opération atomique** signifie que la séquence d'instructions est **indivisible** ⇨ elle est **garantie** d'être exécutée en un seul bloc.

- Les instructions machines ne sont pas forcément atomiques.
- Le mécanisme d'atomicité est nécessaire dans le cas d'accès concurrents à des données.
- Exemples :
  - Atomicité des transactions dans une base de données
  - Atomicité au niveau des opérations du système de fichiers
  - Concurrence dans les systèmes d'exploitation, etc.

Thread-Safe

- Dans un contexte concurrent, toute fonction non thread-safe doit être protégée par un mécanisme approprié (ex: verrou)
- La plupart des fonctions de la librairie POSIX Threads sont thread-safe; celles qui ne le sont pas sont listées dans le manuel (man pthreads)

> Une fonction est dite **thread-safe** (ou **MT-safe**) si elle peut être appelée simultanément par plusieurs threads et qu'elle retourne toujours le même résultat.

Code Thread-safe

- Fonction thread-safe : bloque indéfiniment si elle s'appelle elle-même (dû au mutex; cf. chapitre suivant) :

```
void f() {
    mutex_lock();
    ...
    // function body
    ...
    mutex_unlock();
}
```

Ressource critique

- Exemples de ressources critiques :
  - Une variable globale (ressource logique) accédée en lecture et en écriture par plusieurs threads
  - Une ressource physique (périphérique) accédée par plusieurs threads.

Une **ressource critique** est une ressource non partageable et accédée par plusieurs threads.

Ressource critique

```
int iterations = 1000000;
int global = 0;
void *my_thread(void *arg) {
  for (int i = 0; i < iterations; i++) {
    global++;
  }
  printf("Fin du thread %d\n", (int)arg);
  return NULL;
}

int main(void) {
  pthread_t thread1, thread2;
  if (pthread_create(&thread1, NULL, my_thread, (void *)1) == 0) {
    if (pthread_create(&thread2, NULL, my_thread, (void *)2) == 0) {
      pthread_join(thread1, NULL);
      pthread_join(thread2, NULL);
      printf("Fin des thread: global = %d (%d)\n", global, 2*iterations);
      return EXIT_SUCCESS;
    }
    printf(stderr, "main: Erreur pthread_create\n");
  }
  printf(stderr, "main: Erreur pthread_create\n");
  return EXIT_FAILURE;
}
```

Ressource critique

```
int iterations = 1000000;
int global = 0;
void *my_thread(void *arg) {
  for (int i = 0; i < iterations; i++) {
    global++;
  }
  printf("Fin du thread %d\n", (int)arg);
  return NULL;
}

int main(void) {
  pthread_t thread1, thread2;
  if (pthread_create(&thread1, NULL, my_thread, (void *)1) == 0) {
    if (pthread_create(&thread2, NULL, my_thread, (void *)2) == 0) {
      pthread_join(thread1, NULL);
      pthread_join(thread2, NULL);
      printf("Fin des thread: global = %d (%d)\n", global, 2*iterations);
      return EXIT_SUCCESS;
    }
    fprintf(stderr, "main: Erreur pthread_create\n");
  }
  fprintf(stderr, "main: Erreur pthread_create\n");
  return EXIT_FAILURE;
}
```

Problèmes de la programmation concurrente

- **Interférences** : plusieurs threads concurrents modifient une donnée en même temps ➔ la donnée peut être corrompue

- Exemple : soit une date de naissance dd/mm/yyyy et deux threads concurrents :

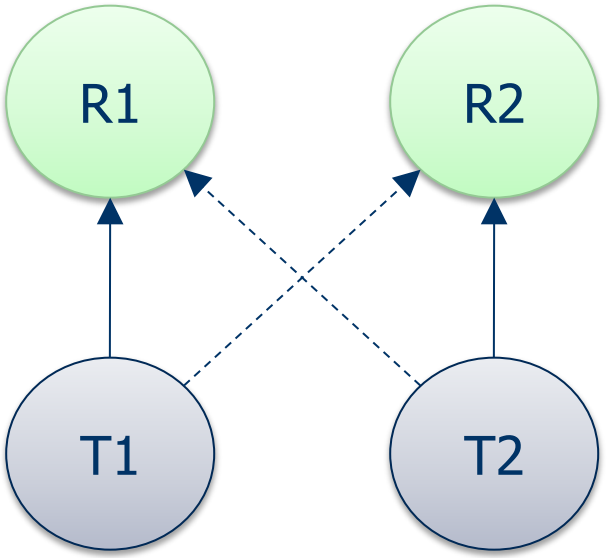
  - Thread1 exécute dd = 28, mm = 12, yyyy = 1969
  - Thread2 exécute dd = 16, mm = 3, yyyy = 1953
  - Après exécution le résultat est :

    jj = 28, mm = 3, yyyy = 1953

  ➔Exclusion mutuelle non garantie !

Problèmes de la programmation concurrente

- **Interblocage (*deadlock*)** : blocage permanent d'un ensemble de threads; survient lorsque un ou plusieurs threads attendent une action d'un autre thread pour continuer
- Exemple ⇨ deux threads concurrents (T1 et T2) s'attendent mutuellement :
  - T1 possède R1 et demande R2
  - T2 possède R2 et demande R1



Section critique

- Une ressource critique doit être exécutée en exclusion mutuelle
  - ➔ les accès à la ressource s'excluent mutuellement
- L'exclusion mutuelle doit être assurée dans une section critique
- L'accès à une section critique se fait par un algorithme d'exclusion mutuelle en deux parties :
  - Protocole d'entrée
  - Protocole de sortie

Une section de code accédant à des ressources partagées et ne devant pas être exécutée par un autre thread est appelée **section critique**.

Modèle

- Modèle de l'exclusion mutuelle pour un thread

```
void *thread(void *arg) {
        …
        // Instructions
        …
        PROTOCOLE D'ENTREE (prélude)
            SECTION CRITIQUE  // Accès à la ressource critique
        PROTOCOLE DE SORTIE (postlude)
        …
        // Instructions
        …
}
```

Exemple

- Modèle de l'exclusion mutuelle pour un thread

```
void *func1(void *arg) {
    for (int i = 0; i < COUNT; i++) {
        Entrée en section critique
        n++;        // SECTION CRITIQUE
        Sortie de la section critique
    }
    printf("Thread 1 terminated.\n");
    return NULL;
}

void *func2(void *arg) {
    for (int i = 0; i < COUNT; i++) {
        Entrée en section critique
        n++;        // SECTION CRITIQUE
        Sortie de la section critique
    }
    printf("Thread 2 terminated.\n");
    return NULL;
}
```