

# Concurrence et non-déterminisme

F. Gluck, V. Pilloux

Version 0.5

# Ordre des opérations

- Soit les threads A et B suivants s'exécutant de manière concurrente, 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 de manière concurrente, 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 ?

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 de manière concurrente, 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 ?

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 de manière concurrente, 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 ?

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`

# Quiz

```
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;
}
```

Quelle date sera  
affichée à la sortie  
du programme ?

# Concurrence et non déterminisme

- Selon **l'ordre des opérations**, la donnée peut être corrompue
- Dans l'exemple précédent, si l'exécution temporelle est:

- 1) `day = 16;`
- 2) `day = 28;`
- 3) `month = 12;`
- 4) `year = 1963;`
- 5) `month = 3;`
- 6) `year = 1953;`

- Après exécution le résultat sera:

`day = 28, month = 3, year = 1953`

- ⇒ L'entrelacement des instructions rend le **résultat non déterministe** !

# Quiz 2

```
#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;
}
```

Quelle sera la valeur de `n` à la fin du programme ?



# Exemple d'exécution

Résultat de plusieurs exécutions :

```
Thread 1 terminated.  
Thread 2 terminated.  
n = 10313  
  
Thread 1 terminated.  
Thread 2 terminated.  
n = 16595  
  
Thread 1 terminated.  
Thread 2 terminated.  
n = 16344  
  
Thread 2 terminated.  
Thread 1 terminated.  
n = 10404  
  
Thread 2 terminated.  
Thread 1 terminated.  
n = 10203  
  
Thread 1 terminated.  
Thread 2 terminated.  
n = 16687  
  
Thread 2 terminated.  
Thread 1 terminated.  
n = 11205  
...
```

# Exemple d'exécution

Résultat de plusieurs exécutions :

Pourquoi  $n$  ne vaut pas 20000 ?

Thread 1 terminated.  
Thread 2 terminated.  
 $n = 10313$

Thread 1 terminated.  
Thread 2 terminated.  
 $n = 16595$

Thread 1 terminated.  
Thread 2 terminated.  
 $n = 16344$

Thread 2 terminated.  
Thread 1 terminated.  
 $n = 10404$

Thread 2 terminated.  
Thread 1 terminated.  
 $n = 10203$

Thread 1 terminated.  
Thread 2 terminated.  
 $n = 16687$

Thread 2 terminated.  
Thread 1 terminated.  
 $n = 11205$

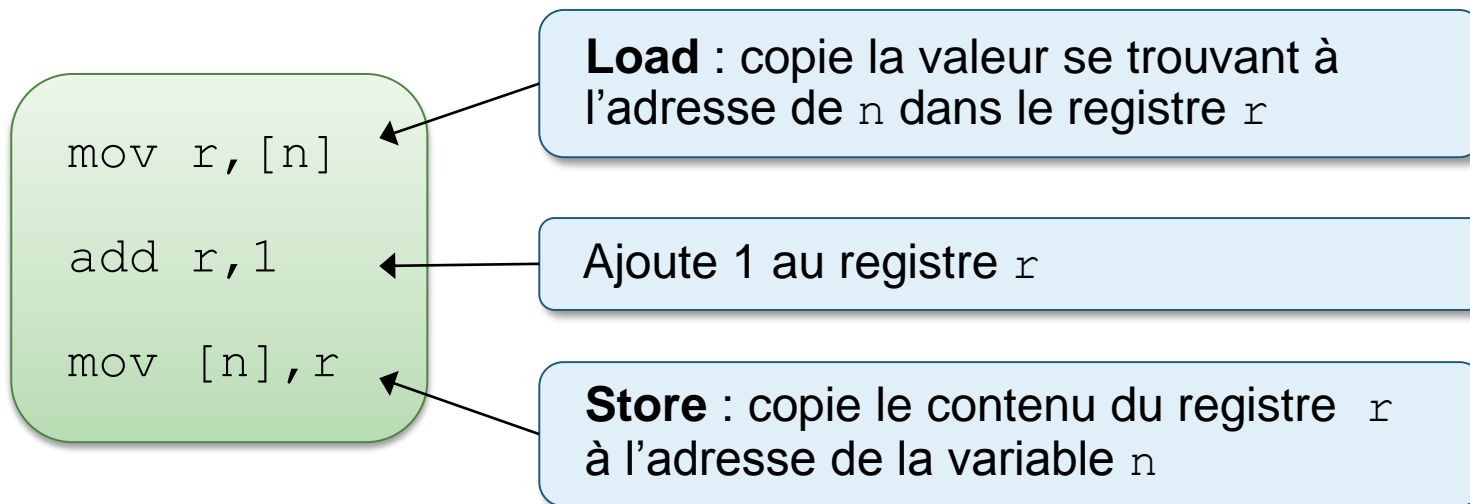
...

# Opération divisible

- On pense obtenir 20000, mais ce n'est le cas.
- Que signifie l'instruction `n++` ?

# Opération divisible


- On pense obtenir 20000, mais ce n'est le cas.
- Que signifie l'instruction `n++` ?
- Cette simple opération C d'incrément est réellement composée de 3 instructions en langage machine :



# Entrelacement

- Soit  $n = 10$
- Note: **r est un registre**, différent dans le contexte de chaque thread
- L'opération  $n++$  peut être interrompue :

Thread 1	Thread 2	
$r \leftarrow 10$		



temps

# Entrelacement

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

Thread 1	Thread 2	
$r \leftarrow 10$		
		Changement de contexte

↓ temps

# Entrelacement

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

Thread 1	Thread 2	
$r \leftarrow 10$		
		Changement de contexte
	$r \leftarrow 10$	

↓ temps

# Entrelacement

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

Thread 1	Thread 2	
$r \leftarrow 10$		
		Changement de contexte
	$r \leftarrow 10$	
	$r \leftarrow r + 1$	

↓ temps



# Entrelacement

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

Thread 1	Thread 2	
$r \leftarrow 10$		
		Changement de contexte
	$r \leftarrow 10$	
	$r \leftarrow r + 1$	
	$n \leftarrow r \ (n = 11)$	

↓ temps

# Entrelacement

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

Thread 1	Thread 2	
$r \leftarrow 10$		
		Changement de contexte
	$r \leftarrow 10$	
	$r \leftarrow r + 1$	
	$n \leftarrow r \text{ (} n = 11 \text{)}$	
		Changement de contexte

↓ temps

# Entrelacement

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

Thread 1	Thread 2	
$r \leftarrow 10$		
		Changement de contexte
	$r \leftarrow 10$	
	$r \leftarrow r + 1$	
	$n \leftarrow r \ (n = 11)$	
		Changement de contexte
$r \leftarrow r + 1$		

↓ temps

# Entrelacement

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

Thread 1	Thread 2	
$r \leftarrow 10$		
		Changement de contexte
	$r \leftarrow 10$	
	$r \leftarrow r + 1$	
	$n \leftarrow r \text{ (} n = 11 \text{)}$	
		Changement de contexte
$r \leftarrow r + 1$		
$n \leftarrow r$		

↓ temps

# Entrelacement

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

Thread 1	Thread 2	
$r \leftarrow 10$		
		Changement de contexte
	$r \leftarrow 10$	
	$r \leftarrow r + 1$	
	$n \leftarrow r \text{ (} n = 11 \text{)}$	
		Changement de contexte
$r \leftarrow r + 1$		
$n \leftarrow r$		
		$n$ vaut 11 et non 12 !

↓ temps

# Situation de concurrence

- Problèmes dûs à la **compétition** pour l'accès aux ressources partagées :
  - Comportement **non déterministe** !
  - Situation de concurrence (*race condition*)

Une **situation de concurrence** est une situation où plusieurs threads lisent et écrivent une ressource partagée et le résultat final **dépend** du **timing** de l'exécution.

# Propriétés

- Atomicité
- Réentrance
- Thread-safety

# Atomicité

- L'exemple d'exécution précédent montre que la variable  $n$  ne peut pas être utilisée comme compteur.
- Solution ?



# 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 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**.
- Solution : « protéger » la section critique afin qu'elle ne soit pas accédée par plusieurs threads (à l'aide d'un verrou par exemple).

# Atomicité

Une **opération atomique** signifie que la séquence d'instructions la composant apparaît comme **indivisible**  
⇒ **garantie** d'être exécutée en un seul bloc.

- A quelques exception près, le code C n'est pas atomique
- Même les **instructions machines** ne sont **pas toujours atomiques** (CISC *Complex Instruction Set Computer*).
- 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.

# Réentrance

Une fonction est dite **réentrante** si elle peut être interrompue au milieu de son exécution puis appelée à nouveau (*réentrée*) **avant** que sa première exécution ne se termine.

- Réentrant réfère à **un seul flot d'exécution**  $\Rightarrow$  concept datant d'avant les systèmes multi-threadés. Exemples : fonction récursive, fonction interrompue par une interruption matérielle.
- Une fonction réentrante :
  - ne devrait pas référencer de données statiques ou globales
  - ne doit pas retourner de pointeur sur des données statiques ou globales
  - ne doit pas appeler de fonction non-réentrante

# Réentrance (2)

Comment modifier une fonction non-réentrante pour qu'elle devienne réentrante ?

- Dans la plupart des cas, rendre une fonction non réentrante réentrante implique de modifier son interface.
- Il peut être impossible de rendre une fonction non réentrante thread-safe (appelable dans un contexte multi-threadé).
- Certaines fonctions de la librairie C ne sont pas réentrantes : `strtok`, `crypt`, etc.

Dans le doute, consulter le manuel (`man`) !

- Certaines fonctions existent en versions réentrantes ; généralement postfixées par le suffixe `_r`.

Exemples : `strtok_r`, `crypt_r`

# Thread-safe

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.

- Dans un contexte concurrent, toute fonction non *thread-safe* doit être protégée (avec un verrou par exemple).
- La plupart des fonctions de la librairie POSIX Threads sont thread-safe. Celles ne l'étant pas sont listées dans le manuel (`man pthreads`).

# Réentrant vs thread-safe

- Réentrant  $\neq$  thread-safe.
- Une fonction réentrante n'est **pas forcément** thread-safe.
- Une fonction thread-safe n'est **pas forcément** réentrante.
- Une fonction non réentrante ne **doit pas** être appelée par plusieurs threads !

# Exemple de code réentrant

## Fonction réentrante ...

```
int t;

void swap(int *x, int *y) {
    int s = t; // sauve la variable globale (atomique)
    t = *x;
    *x = *y;
    f();      appel par interruption -> OK (réentrant)
    *y = t;
    t = s; // restore la variable globale (atomique)
}

void f() {
    int x = 1, y = 2;
    swap(&x, &y);
}
```

# Exemple de code réentrant

... **mais pas thread-safe** : ne garantit pas un état cohérent de la variable `t` lors d'une exécution multithreadée!

```
int t;

void swap(int *x, int *y) {
    int s = t; // sauve la variable globale (atomique)
    t = *x;
    *x = *y;
    // autre thread appelant swap():
    *y = t; // problème -> corruption possible de t!
    t = s; // restore la variable globale (atomique)
}

void f() {
    int x = 1, y = 2;
    swap(&x, &y);
}
```



# Exemple de code thread-safe

**Fonction thread-safe mais pas réentrante** : bloque indéfiniment **si elle s'appelle elle-même** (dû au verrouillage du mutex; cf. chapitre suivant) :

```
void f() {  
    lock(&mutex);    // start of critical section  
    ...  
    // function body  
    f();    // bloquant  
    ...  
    unlock(&mutex); // end of critical section  
}
```

# Thread-safe et librairie C

- Important de lire la documentation (`man`) pour déterminer si une fonction est thread-safe ou réentrante.
- La majorité des fonctions de la glibc est thread-safe.
- **Toujours** consulter le manuel (`man`) en cas de doute !
- Les opérations sur les streams sont thread-safe\* (la glibc implémente un mécanisme de lock interne).

\* [https://www.gnu.org/software/libc/manual/html\\_node/Streams-and-Threads.html](https://www.gnu.org/software/libc/manual/html_node/Streams-and-Threads.html)

# Ressource critique

Une **ressource critique** est une ressource commune, qui est accédée par plusieurs threads en concurrence.

- Exemples :
  - Variable globale (ressource logique) accédée en lecture et en écriture par plusieurs threads.
  - Ressource physique (périphérique) accédée par plusieurs threads.

# Ressource critique : exemple

```
#define CHECK_ERR(expr,msg) \
if (expr) { perror(msg); exit(EXIT_FAILURE); }

const int N = 2;
const int iterations = 1000000;
int global;

void *my_thread(void *arg) {
    for (int i = 0; i < iterations; i++)
        global++;
    return NULL;
}

int main(void) {
    pthread_t t[N];
    for (int i = 0; i < N; i++)
        CHECK_ERR(pthread_create(&t[i], NULL, my_thread, NULL),
            "pthread_create");

    for (int i = 0; i < N; i++)
        CHECK_ERR(pthread_join(t[i], NULL), "pthread_join");

    printf("global = %d (should be %d)\n", global, N*iterations);
    return EXIT_SUCCESS;
}
```

Y-a-t-il une  
ressource critique ?

# Ressource critique : exemple

```
#define CHECK_ERR(expr,msg) \
if (expr) { perror(msg); exit(EXIT_FAILURE); }

const int N = 2;
const int iterations = 1000000;
int global;

void *my_thread(void *arg) {
    for (int i = 0; i < iterations; i++)
        global++;
    return NULL;
}

int main(void) {
    pthread_t t[N];
    for (int i = 0; i < N; i++)
        CHECK_ERR(pthread_create(&t[i], NULL, my_thread, NULL),
            "pthread_create");

    for (int i = 0; i < N; i++)
        CHECK_ERR(pthread_join(t[i], NULL), "pthread_join");

    printf("global = %d (should be %d)\n", global, N*iterations);
    return EXIT_SUCCESS;
}
```

# Section critique

Une section de code accédant à des variables partagées (ou autres ressources partagées) et qui doit être accédée atomiquement est appelé une **section critique**.

# Section critique

```
#define CHECK_ERR(expr,msg) \
if (expr) { perror(msg); exit(EXIT_FAILURE); }

const int N = 2;
const int iterations = 1000000;
int global;

void *my_thread(void *arg) {
    for (int i = 0; i < iterations; i++)
        global++;
    return NULL;
}

int main(void) {
    pthread_t t[N];
    for (int i = 0; i < N; i++)
        CHECK_ERR(pthread_create(&t[i], NULL, my_thread, NULL),
            "pthread_create");

    for (int i = 0; i < N; i++)
        CHECK_ERR(pthread_join(t[i], NULL), "pthread_join");

    printf("global = %d (should be %d)\n", global, N*iterations);
    return EXIT_SUCCESS;
}
```

Où devrait-on avoir  
une section critique ?

# Section critique

```
#define CHECK_ERR(expr,msg) \
if (expr) { perror(msg); exit(EXIT_FAILURE); }

const int N = 2;
const int iterations = 1000000;
int global;

void *my_thread(void *arg) {
    for (int i = 0; i < iterations; i++)
        global++;
    return NULL;
}

int main(void) {
    pthread_t t[N];
    for (int i = 0; i < N; i++)
        CHECK_ERR(pthread_create(&t[i], NULL, my_thread, NULL),
            "pthread_create");

    for (int i = 0; i < N; i++)
        CHECK_ERR(pthread_join(t[i], NULL), "pthread_join");

    printf("global = %d (should be %d)\n", global, N*iterations);
    return EXIT_SUCCESS;
}
```



# Section critique : accès

- Une ressource critique doit être accédée en **exclusion mutuelle**  
⇒ les accès à la ressource s'excluent mutuellement.

**L'exclusion mutuelle doit être assurée dans une section critique.**