

# Détection dynamique de conditions de course

Félix-Antoine Ouellet

Université de Sherbrooke

6 novembre 2014

1 Motivation

2 Arrivé-avant

3 Ensemble de verrous

4 Comparaison

5 Conclusion

# Plan

- 1 Motivation
- 2 Arrivé-avant
- 3 Ensemble de verrous
- 4 Comparaison
- 5 Conclusion

# Condition de course

## Définition

Situation se produisant quand 2 *threads* accèdent à la même structure partagée sans contraintes d'ordonnancement et qu'un de ces accès est une écriture.

# Condition de course

## Exemple - Trivial

```
int main() {  
    int X = 0;  
    std::thread T([&]() { X = 42; });  
    X = 43;  
    T.join();  
}
```

Que vaut X à la fin du programme?

# Condition de course

## Exemple - Moins trivial

```
Singleton* Singleton::getInstance() {  
    if (m_Instance == nullptr) {  
        std::lock_guard<std::mutex> Lock(m_Mutex);  
        {  
            if (m_Instance == nullptr) {  
                m_Instance = new Singleton;  
            }  
        }  
    }  
    return m_Instance;  
}
```

# Plan

- 1 Motivation
- 2 Arrivé-avant
  - Idée
  - Concepts de base
  - Algorithme
- 3 Ensemble de verrous
- 4 Comparaison
- 5 Conclusion

# Idée

Un programme parallèle sans condition de course ne comporte que des accès ordonnancés à des structures partagées



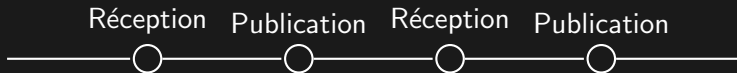
# Opérations de synchronisation

## Théorie

- Publication: Rend publique de l'information produite par le *thread*
- Réception: Lecture d'une information publique

# Opérations de synchronisation

## Pratique



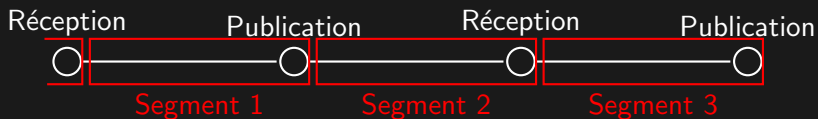
# Segments

## Théorie

Suite d'opérations effectuées par un *thread* se terminant par une opération de synchronisation

# Segments

## Pratique



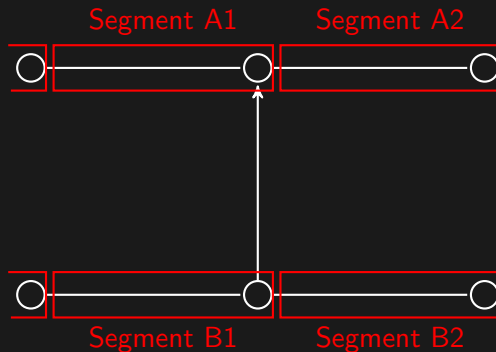
# Ordonnancement des segments

## Théorie

- Un ordre partiel peut être établi en fonction des opérations de synchronisation
- Dénuté par l'opérateur  $\prec$

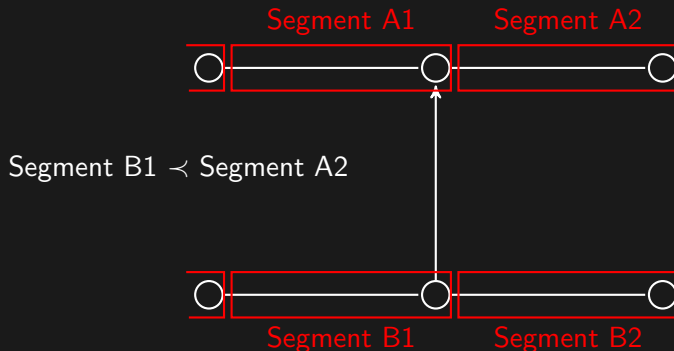
# Ordonnancement des segments

## Pratique



# Ordonnement des segments

## Pratique



# Horloge vectorielle

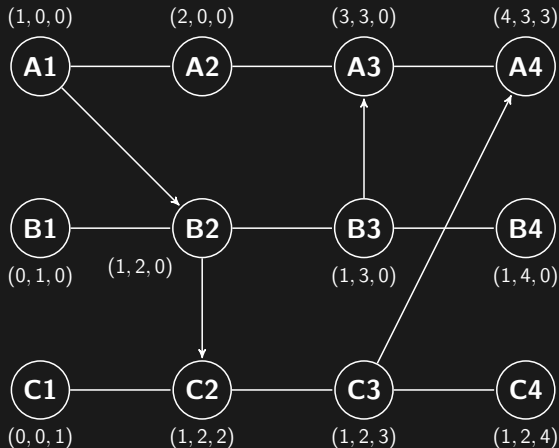
## Théorie

Structure permettant d'effectuer un ordonnancement partiel des événements dans un système parallèle



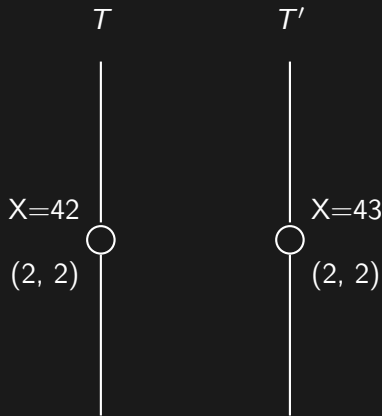
# Horloge vectorielle

## Pratique



## Conditions de course

Deux accès mémoire dont au moins un est une écriture et qui pourraient être exécutées simultanément sans connaissance des manipulations effectuées par l'autre *thread*



# Algorithme

## Équations

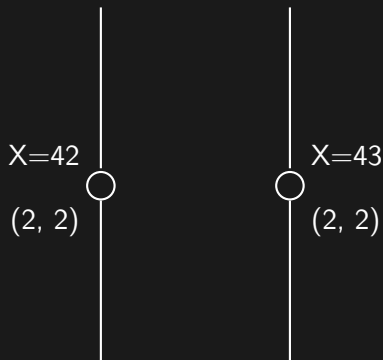
Il y a une condition de course entre un segment  $S$  et un segment  $S'$  si:

1.  $V_S(T) \geq V_{S'}(T)$  et  $V_{S'}(T') \geq V_S(T')$
2.  $[R_S \cup W_S] \cap W_{S'} \neq \emptyset$  ou  $[R_{S'} \cup W_{S'}] \cap W_S \neq \emptyset$

# Algorithme

## Pratique

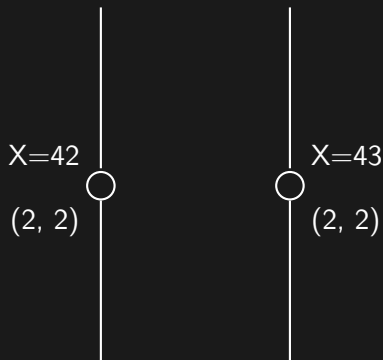
$$\checkmark \quad V_S(T) \geq V_{S'}(T)$$



# Algorithme

## Pratique

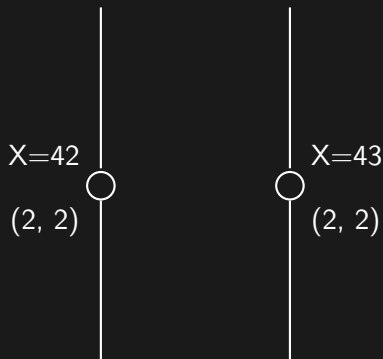
- ✓  $V_S(T) \geq V_{S'}(T)$
- ✓  $V_{S'}(T') \geq V_S(T')$



# Algorithme

## Pratique

- ✓  $V_S(T) \geq V_{S'}(T)$
- ✓  $V_{S'}(T') \geq V_S(T')$
- ✓  $[R_{S'} \cup W_{S'}] \cap W_S \neq \emptyset$



# Plan

- 1 Motivation
- 2 Arrivé-avant
- 3 Ensemble de verrous**
  - Idée
  - Algorithme
- 4 Comparaison
- 5 Conclusion

# Idée

Un programme parallèle sans condition de course respecte toujours une saine discipline de verrouillage des structures partagées



# Algorithme

## Ébauche

But: S'assurer que toute structure partagée soit protégée par un verrou



# Algorithme

## Ébauche

```
Mutex1.lock();  
v = v + 1;  
Mutex1.unlock();  
Mutex2.lock();  
v = v + 1;  
Mutex2.unlock();
```



Verrous	$C(v)$
$\{\}$	$\{\text{Mutex1, Mutex2}\}$

# Algorithme

## Ébauche

```
Mutex1.lock();  
v = v + 1;  
Mutex1.unlock();  
Mutex2.lock();  
v = v + 1;  
Mutex2.unlock();
```



Verrous	$C(v)$
$\{\}$	$\{\text{Mutex1, Mutex2}\}$
$\{\text{Mutex1}\}$	$\{\text{Mutex1, Mutex2}\}$

# Algorithme

## Ébauche

```
Mutex1.lock();
v = v + 1;
Mutex1.unlock();
Mutex2.lock();
v = v + 1;
Mutex2.unlock();
```



Verrous	$C(v)$
$\{\}$	$\{\text{Mutex1, Mutex2}\}$
$\{\text{Mutex1}\}$	$\{\text{Mutex1, Mutex2}\}$
$\{\text{Mutex1}\}$	$\{\text{Mutex1}\}$

# Algorithme

## Ébauche

```
Mutex1.lock();
v = v + 1;
Mutex1.unlock();
Mutex2.lock();
v = v + 1;
Mutex2.unlock();
```

⇐

Verrous	$C(v)$
$\{\}$	$\{\text{Mutex1, Mutex2}\}$
$\{\text{Mutex1}\}$	$\{\text{Mutex1, Mutex2}\}$
$\{\text{Mutex1}\}$	$\{\text{Mutex1}\}$
$\{\}$	$\{\text{Mutex1}\}$

# Algorithme

## Ébauche

```
Mutex1.lock();
v = v + 1;
Mutex1.unlock();
Mutex2.lock();
v = v + 1;
Mutex2.unlock();
```



Verrous	$C(v)$
$\{\}$	$\{\text{Mutex1, Mutex2}\}$
$\{\text{Mutex1}\}$	$\{\text{Mutex1, Mutex2}\}$
$\{\text{Mutex1}\}$	$\{\text{Mutex1}\}$
$\{\}$	$\{\text{Mutex1}\}$
$\{\text{Mutex2}\}$	$\{\text{Mutex1}\}$

# Algorithme

## Ébauche

```
Mutex1.lock();
v = v + 1;
Mutex1.unlock();
Mutex2.lock();
v = v + 1;
Mutex2.unlock();
```



Verrous	$C(v)$
$\{\}$	$\{\text{Mutex1}, \text{Mutex2}\}$
$\{\text{Mutex1}\}$	$\{\text{Mutex1}, \text{Mutex2}\}$
$\{\text{Mutex1}\}$	$\{\text{Mutex1}\}$
$\{\}$	$\{\text{Mutex1}\}$
$\{\text{Mutex2}\}$	$\{\text{Mutex1}\}$
$\{\text{Mutex2}\}$	$\{\text{Mutex1}\}$



# Algorithme

## Ébauche

```
Mutex1.lock();
v = v + 1;
Mutex1.unlock();
Mutex2.lock();
v = v + 1;
Mutex2.unlock();
```



Verrous	$C(v)$
$\{\}$	$\{\text{Mutex1}, \text{Mutex2}\}$
$\{\text{Mutex1}\}$	$\{\text{Mutex1}, \text{Mutex2}\}$
$\{\text{Mutex1}\}$	$\{\text{Mutex1}\}$
$\{\}$	$\{\text{Mutex1}\}$
$\{\text{Mutex2}\}$	$\{\text{Mutex1}\}$
$\{\text{Mutex2}\}$	$\{\text{Mutex1}\}$
$\{\}$	$\{\}$

# Algorithme

## Ébauche

Trois problèmes de l'algorithme précédent

- Initialisation
- Structure seulement en lecture
- Verrou lecture-écriture

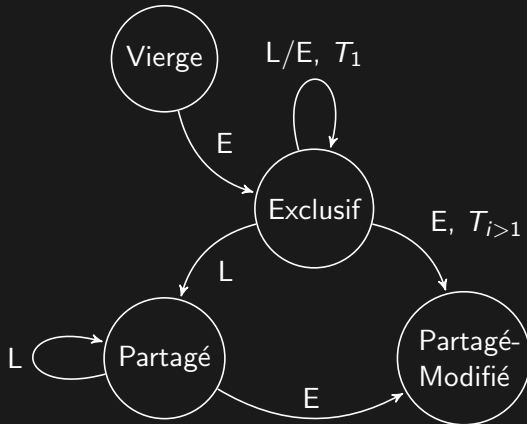
# Algorithme

## Raffinement

$\text{Verrous}(T) = \text{Ensemble de verrous acquis par un } \textit{thread} \ T;$   
 $\text{Verrous.É}(T) = \text{Ensemble de verrous en mode écriture acquis par un } \textit{thread} \ T;$   
 $C(v) = \text{verrous possibles pour une variable } v;$   
**for** *Toute variable partagé*  $v$  **do**  
 |  $C(v) = \text{tous les verrous présents dans l'application}$   
**end**  
**for** *Toute lecture d'une variable partagé*  $v$  **do**  
 |  $C(v) = C(v) \cap \text{Verrous}(T);$   
 | **if**  $C(v) == \{\}$  **then**  
 | | Condition de course détectée  
 | **end**  
**end**  
**for** *Toute écriture d'une variable partagé*  $v$  **do**  
 |  $C(v) = C(v) \cap \text{Verrous.É}(T);$   
 | **if**  $C(v) == \{\}$  **then**  
 | | Condition de course détectée  
 | **end**  
**end**

# Algorithme

## Raffinement



# Plan

- 1 Motivation
- 2 Arrivé-avant
- 3 Ensemble de verrous
- 4 Comparaison
  - Implémentation
  - Conditions de course détectées
- 5 Conclusion

# Implémentation

## Instrumentation

- Recompilation dynamique de binaires
- Fait souvent appel à de la *shadow memory*
- Ne diffère que dans les instructions visés

# Implémentation

## Annotations

Pour aider le processus de détections, les outils offrent souvent la possibilité d'annoter un programme. Par exemple:

### ThreadSanitizer

```
ANNOTATE_IGNORE_WRITES_BEGIN  
ANNOTATE_IGNORE_WRITES_END  
ANNOTATE_CONDVAR_LOCK_WAIT(cv, mu)  
ANNOTATE_BENIGN_RACE(ptr)
```

### Eraser

```
EraserReadLock(lock)  
EraserReadUnLock(lock)  
EraserWriteLock(lock)  
EraserWriteUnLock(lock)
```

## Cas non détecté par arrivé-avant

Thread 1

```
Y += 1;  
Mutex.lock();  
V += 1;  
Mutex.unlock();
```

Thread 2

```
Mutex.lock();  
V += 1;  
Mutex.unlock();  
Y += 1;
```



## Cas non détecté par ensemble de verrous

```
int main() {  
    int X = 0;  
    std::thread T([&]() { X++; });  
    X += 42;  
    T.join();  
}
```

# Résumé

Les faits importants à retenir:

- Arrivé-avant détecte moins de conditions de course que l'ensemble de verrous
- L'ensemble de verrou peut produire des faux positifs contrairement à arrivé-avant
- Les deux algorithmes peuvent produire des faux négatifs

# Plan

- 1 Motivation
- 2 Arrivé-avant
- 3 Ensemble de verrous
- 4 Comparaison
- 5 Conclusion

# Conclusion

La plupart des outils de détection de condition de courses implémentent une variation ou une combinaison des algorithmes présentés.