Détection dynamique de conditions de course

Félix-Antoine Ouellet

Université de Sherbrooke

6 novembre 2014

Plan

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

Example - 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

Example - 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

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

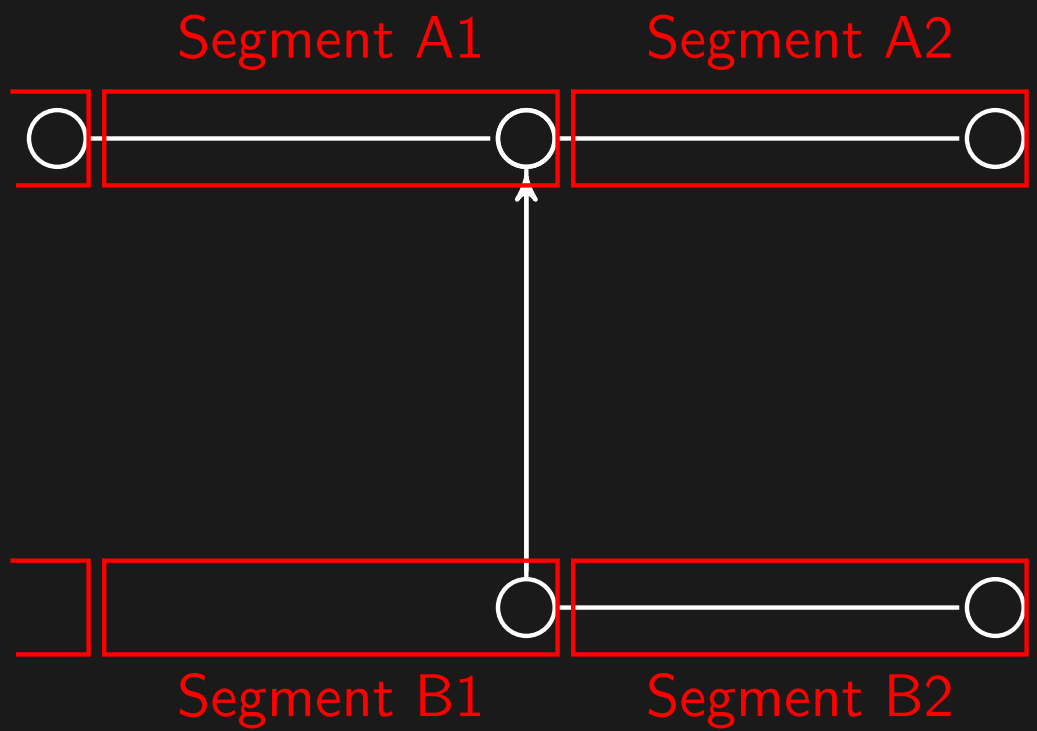
Ordonnancement des segments

Théorie

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

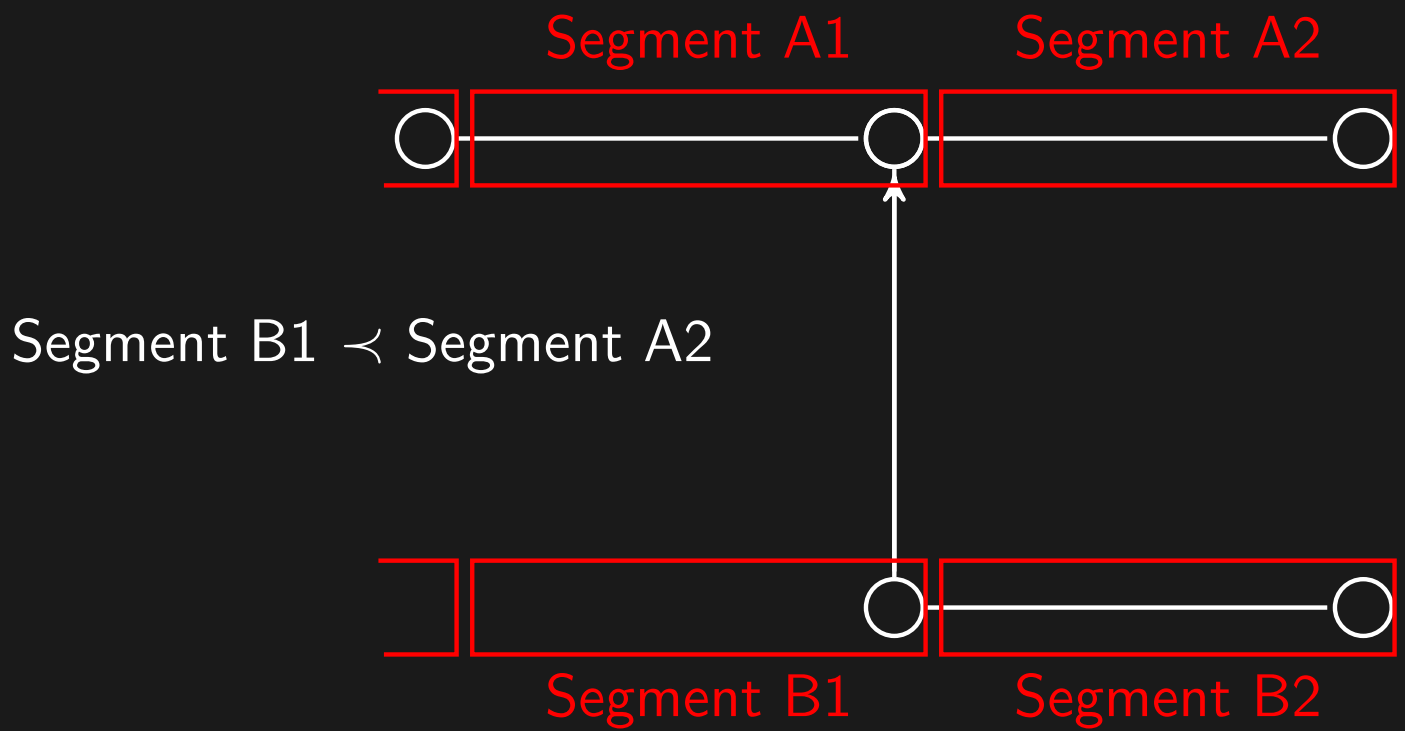
Ordonnancement des segments

Pratique



Ordonnancement des segments

Pratique



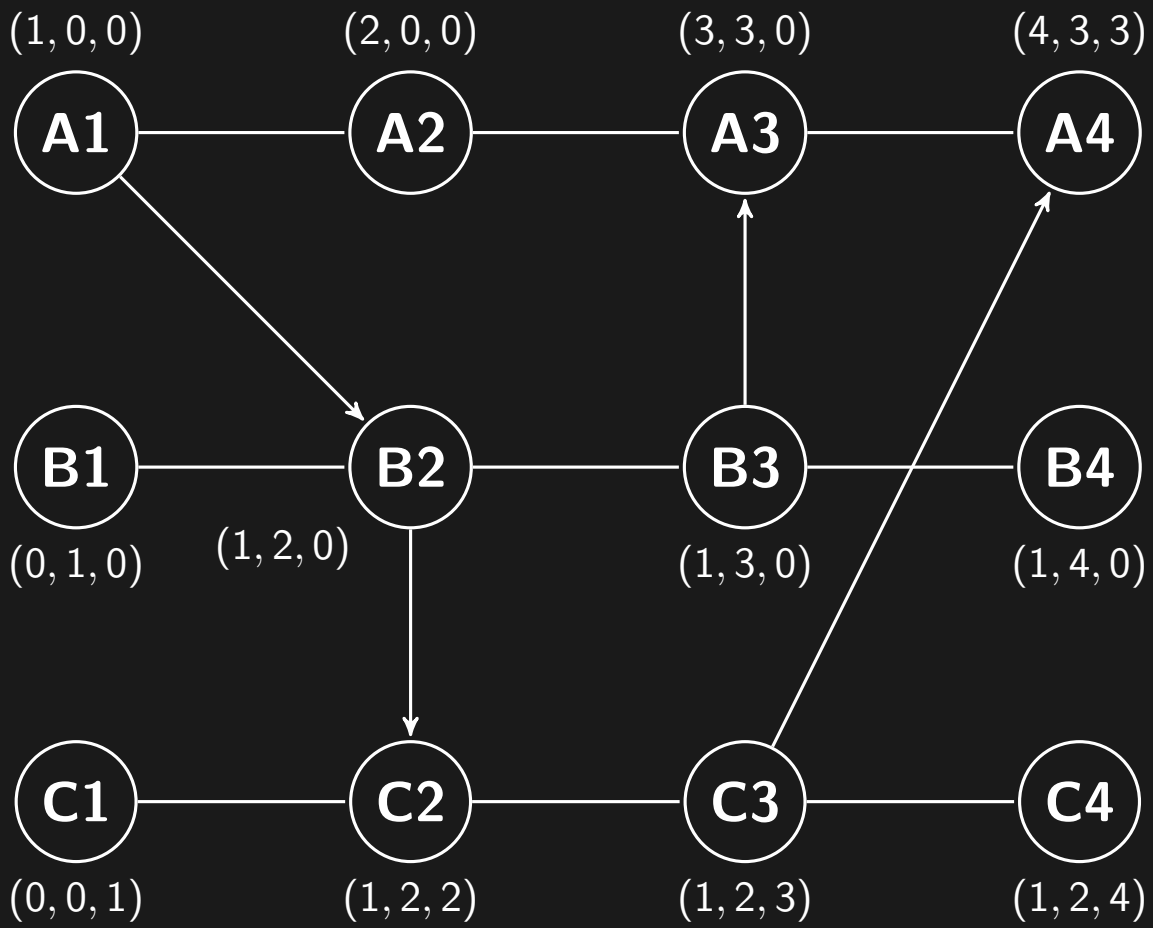
Horloge vectorielle

Théorie

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

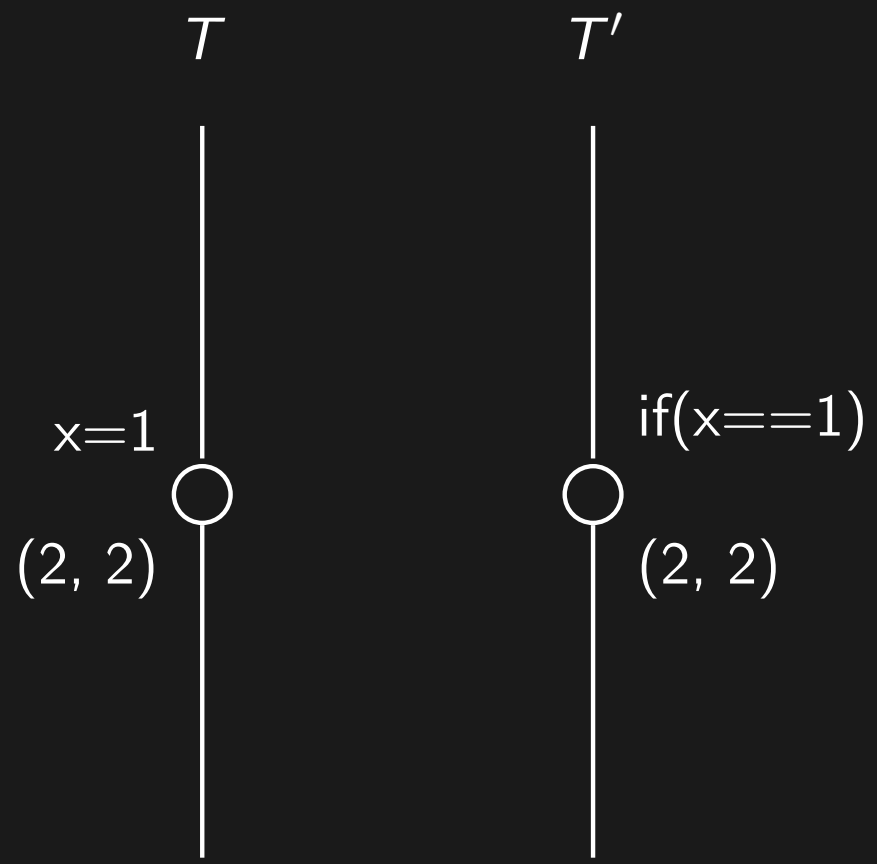
Horloge vectorielle

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

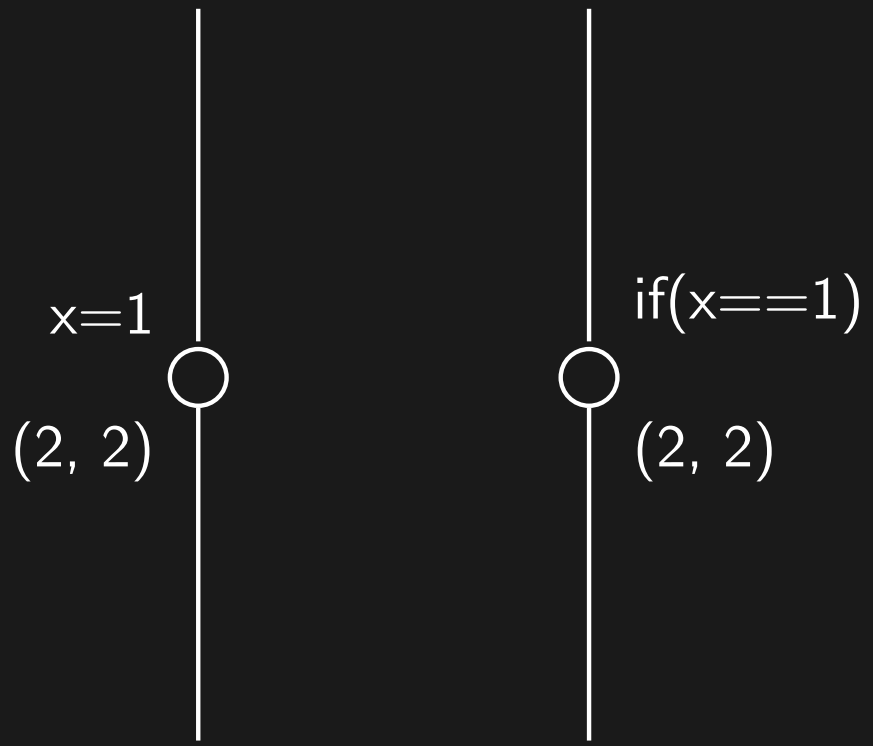
Équations

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 $[R_{S'} \cup W_{S'}] \cap W_S \neq \emptyset$

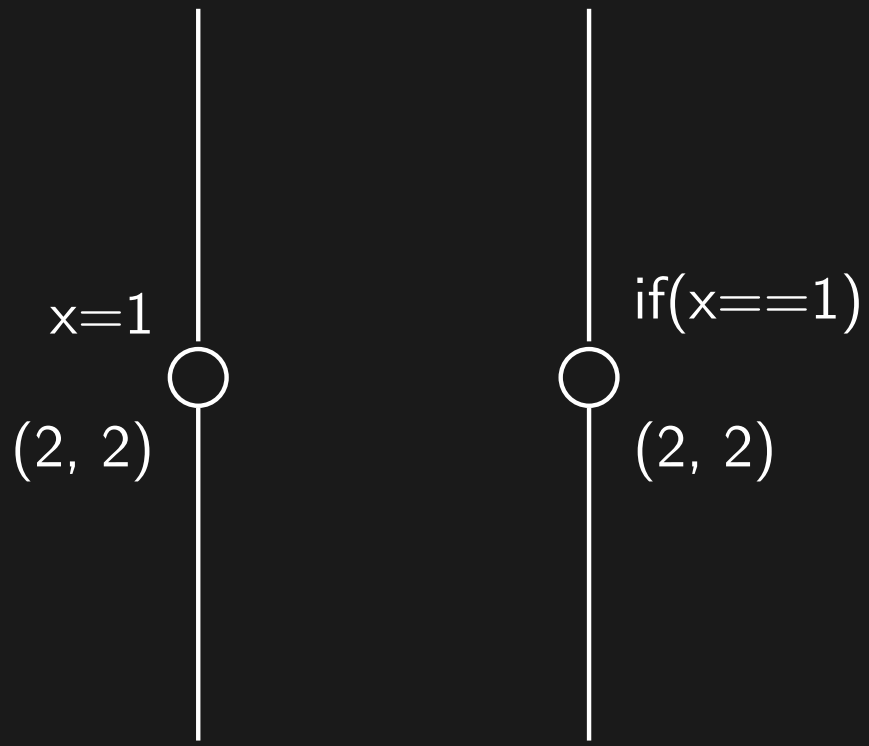
Algorithme
Pratique

✓ $V_S(T) \geq V_{S'}(T)$



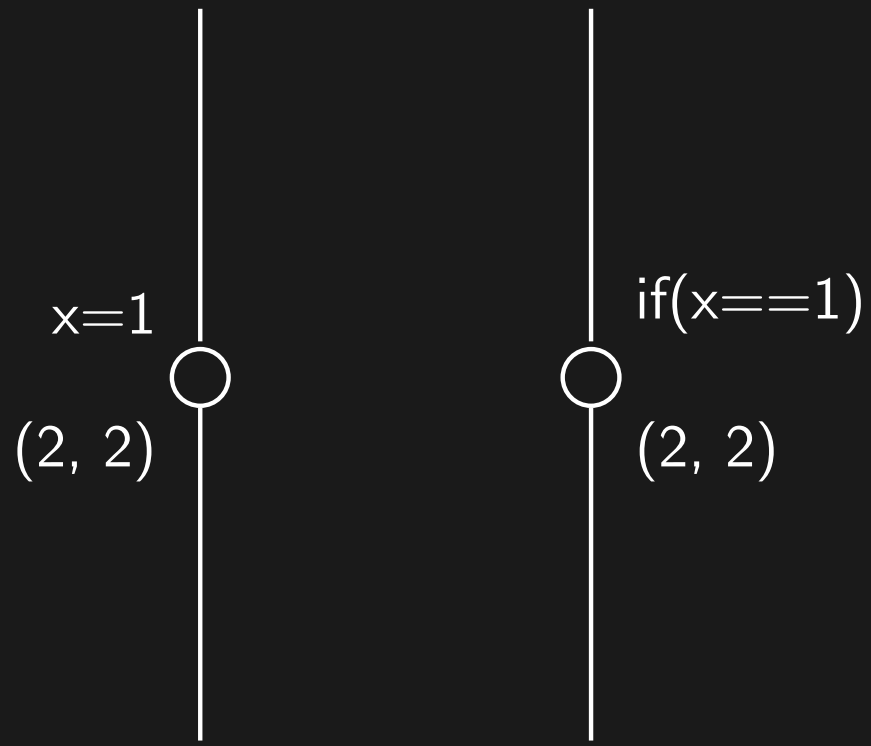
Algorithme
Pratique

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



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

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

Verrous(T) = Ensemble de verrous acquis par un *thread* T;
C(v) = verrous possibles pour une variable v;
for *Toute variable partagé v* **do**
  C(v) = tous les verrous présents dans l'application
end
for *Tout accès à une variable partagé v* **do**
  C(v) = C(v) ∩ Verrous(T);
  if *C(v)* == {} **then**
    Condition de course détectée
  end
end

Algorithme

Ébauche

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

$\Longleftarrow$

Verrous	C(v)
{}	{Mutex1, Mutex2}

Algorithme

Ébauche

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

$\Longleftarrow$

Verrous	C(v)
{}	{Mutex1, Mutex2}
{Mutex1}	{Mutex1, Mutex2}

Algorithme

Ébauche

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

$\Longleftarrow$

Verrous	C(v)
{}	{Mutex1, Mutex2}
{Mutex1}	{Mutex1, Mutex2}
{Mutex1}	{Mutex1}

Algorithme

Ébauche

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

$\Longleftarrow$

Verrous	C(v)
{}	{Mutex1, Mutex2}
{Mutex1}	{Mutex1, Mutex2}
{Mutex1}	{Mutex1}
{}	{Mutex1}

Algorithme

Ébauche

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

$\Longleftarrow$

Verrous	C(v)
{}	{Mutex1, Mutex2}
{Mutex1}	{Mutex1, Mutex2}
{Mutex1}	{Mutex1}
{}	{Mutex1}
{Mutex2}	{Mutex1}

Algorithme

Ébauche

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

$\Leftarrow$

Verrous	C(v)
{}	{Mutex1, Mutex2}
{Mutex1}	{Mutex1, Mutex2}
{Mutex1}	{Mutex1}
{}	{Mutex1}
{Mutex2}	{Mutex1}
{Mutex2}	{Mutex1}

Algorithme

Ébauche

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

$\Longleftarrow$

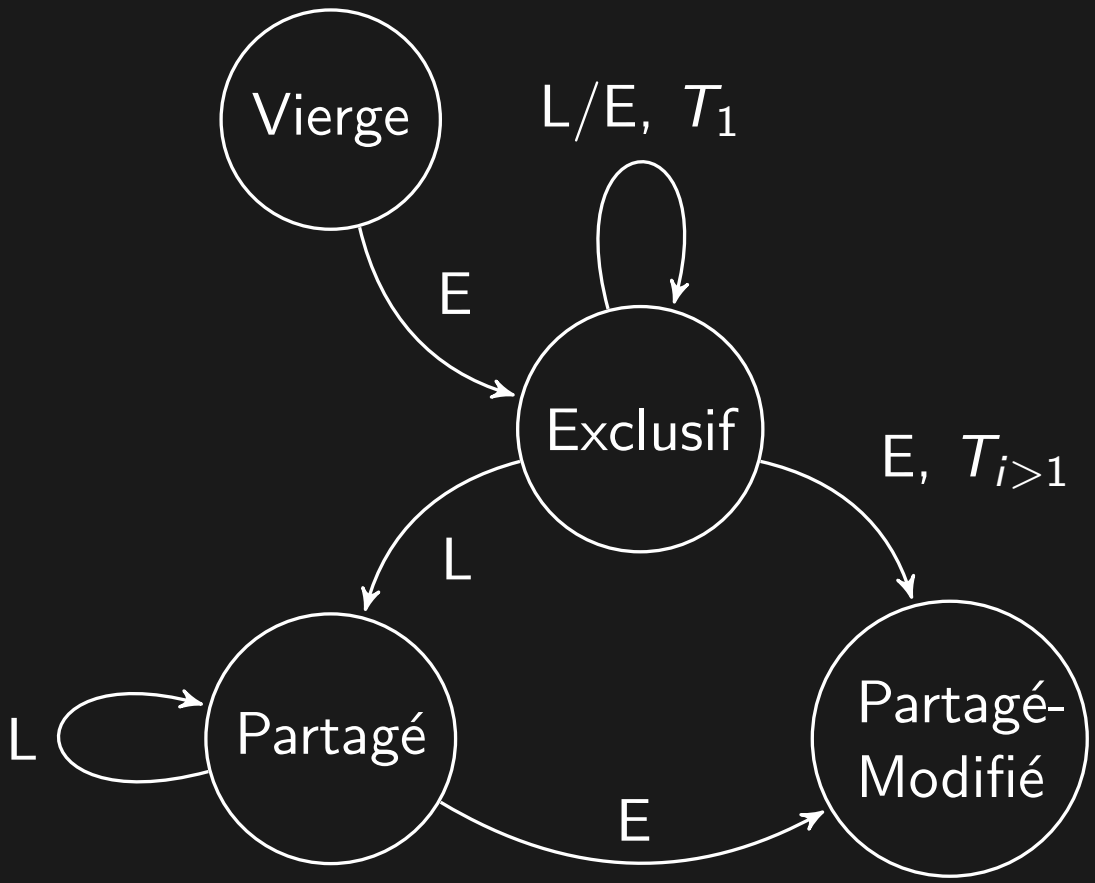
Verrous	C(v)
{}	{Mutex1, Mutex2}
{Mutex1}	{Mutex1, Mutex2}
{Mutex1}	{Mutex1}
{}	{Mutex1}
{Mutex2}	{Mutex1}
{Mutex2}	{Mutex1}
{}	{}

Algorithme
Ébauche

Trois problèmes de l'algorithme précédent

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

Algorithme
Raffinement



Algorithme
Raffinement

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

Plan

Comparaison

Arrivé-avant

- Aucun faux positif

Ensemble de verrous

- Aucun faux négatif

Plan

Conclusion

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