Plan

- Présentation
  - Introduction
  - Historique
- Programmation sous Linux
  - Fichier
  - Processus
  - Communication
  - Threads
  - Synchronisation
- Virtualisation et conteneurisation

Processus légers (threads)

- Le temps de création (et de changement) d'un processus est élevé (… mémoire virtuelle … mémoire cache …).

- Un ***thread*** a son propre compteur ordinal, sa propre pile ... mais n'existe qu'à l'intérieur d'un processus et utilise ses ressources (espace mémoire en particulier).
- Un thread a aussi un état (actif, prêt, bloqué), un bloc de contexte (plus petit !).
- C'est aussi le système d'exploitation qui gère les threads.

Threads POSIX (1)

- Creation

```
int pthread_create(
  pthread_t * thread,
  const pthread_attr_t * attr,
  void * (*start_routine)( void * ),
  void * arg
);
```

- Terminaison sur la fin de `start_routine` ou avec

```
void pthread_exit(
  void * retval
);
```

- Sur Linux, la création de thread se fait via l'appel système `clone()`

Threads POSIX (2)

- Obtenir son identifiant de thread

```
pthread_t pthread_self();
```

- Attente de terminaison (sinon le thread continue d'exister)

```
int pthread_join(
  pthread_t thread,
  void ** retval
);
```

- Détachement (il n'est plus possible de l'attendre)

```
int pthread_detach(
  pthread_t thread
);
```

C++ : std::thread

```
class thread {
public:
  template< class F > explicit thread( F f );

  void join();
  void detach();
  id get_id() const;
  // ...
};

namespace this_thread {
  id get_id();
  void sleep_until( /* ... */ );
  void sleep_for( /* ... */ );
  // ...
}
```

C++20 : std::jthread

Utilisation std::thread

```
void f1( int   n )  { /* ... */ }
void f2( int & n )  { /* ... */ }
struct F3 {
  void operator()() { /* ... */ }
  // ...
};

int main()
{
  int n1{ 5 }, n2{ -2 };
  std::thread t1{ f1, n1 };
  std::thread t2{ f2, std::ref( n2 )};
  F3 f3;
  std::thread t3{ f3 };
  t1.join(); t2.join(); t3.join();
}
```

Démo

GitLab

`cours/progsyst/Thread.cpp`

std::async et std::future

```
int foo()
{
    std::this_thread::sleep_for( 10ms );
    return 42;
}

int main()
{
    std::future< int > f{ std::async( foo )};
    while( f.wait_for( 1ms ) == std::future_status::timeout ) {
        std::cout << "waiting\n";
    }
    std::cout << "got " << f.get() << "\n";
}
```

Démo

GitLab

`cours/progsyst/Async.cpp`

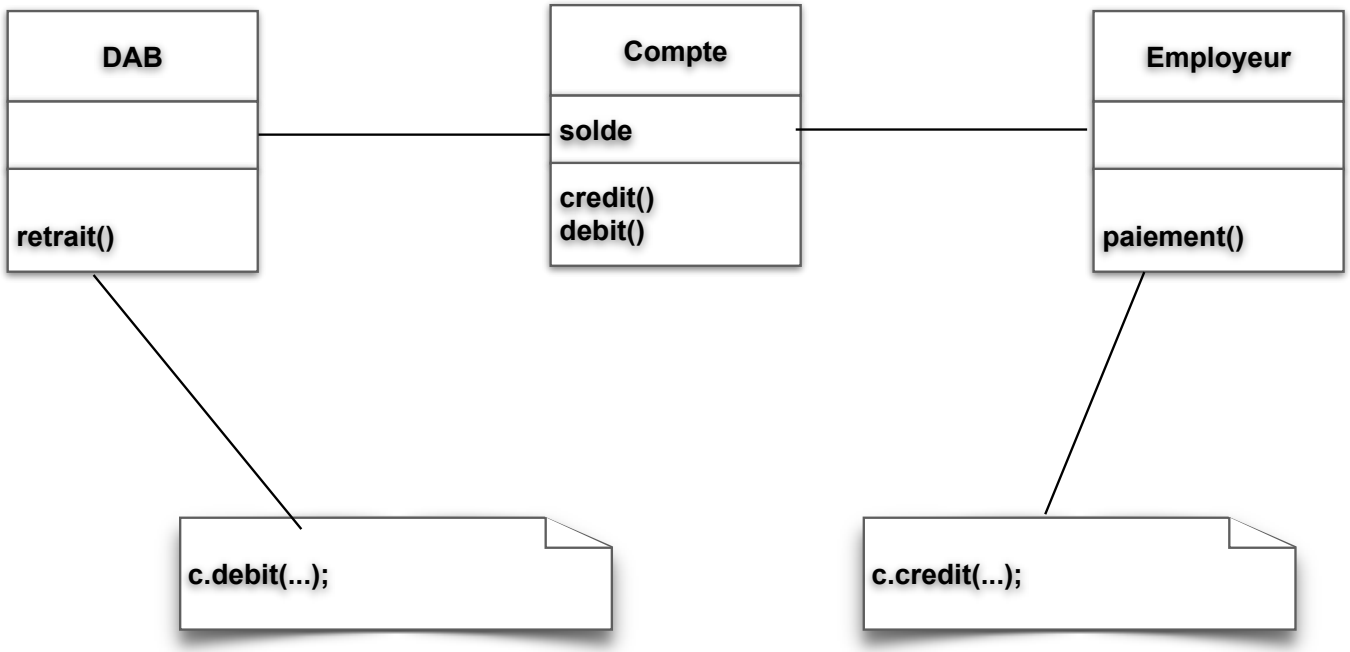
Plan

- Présentation
  - Introduction
  - Historique
- Programmation sous Linux
  - Fichier
  - Processus
  - Communication
  - Threads
  - Synchronisation
- Virtualisation et conteneurisation

Plan

- Présentation
- Programmation sous Unix
  - Fichier
  - Processus
  - Communication
  - Threads
  - Synchronisation
    - Problématique
    - Synchronisation de threads
    - Synchronisation de processus
- Virtualisation et conteneurisation

Problème de la variable partagée



Version incorrecte

```
class Compte {
public:
  void credit( double montant ) {
    solde_ += montant;
  }

  void debit( double montant ) {
    solde_ -= montant;
  }

private:
  double solde_;
}
```

Version incorrecte

```
class Compte {
public:
  void credit( double montant ) {
    solde_ += montant;
  }

  void debit( double montant ) {
    solde_ -= montant;
  }

private:
  double solde_;
}
```

1. Chargement de l'attribut `solde` (en mémoire) dans un registre du processeur

Version incorrecte

```
class Compte {
public:
  void credit( double montant ) {
    solde_ += montant;
  }

  void debit( double montant ) {
    solde_ -= montant;
  }

private:
  double solde_;
}
```

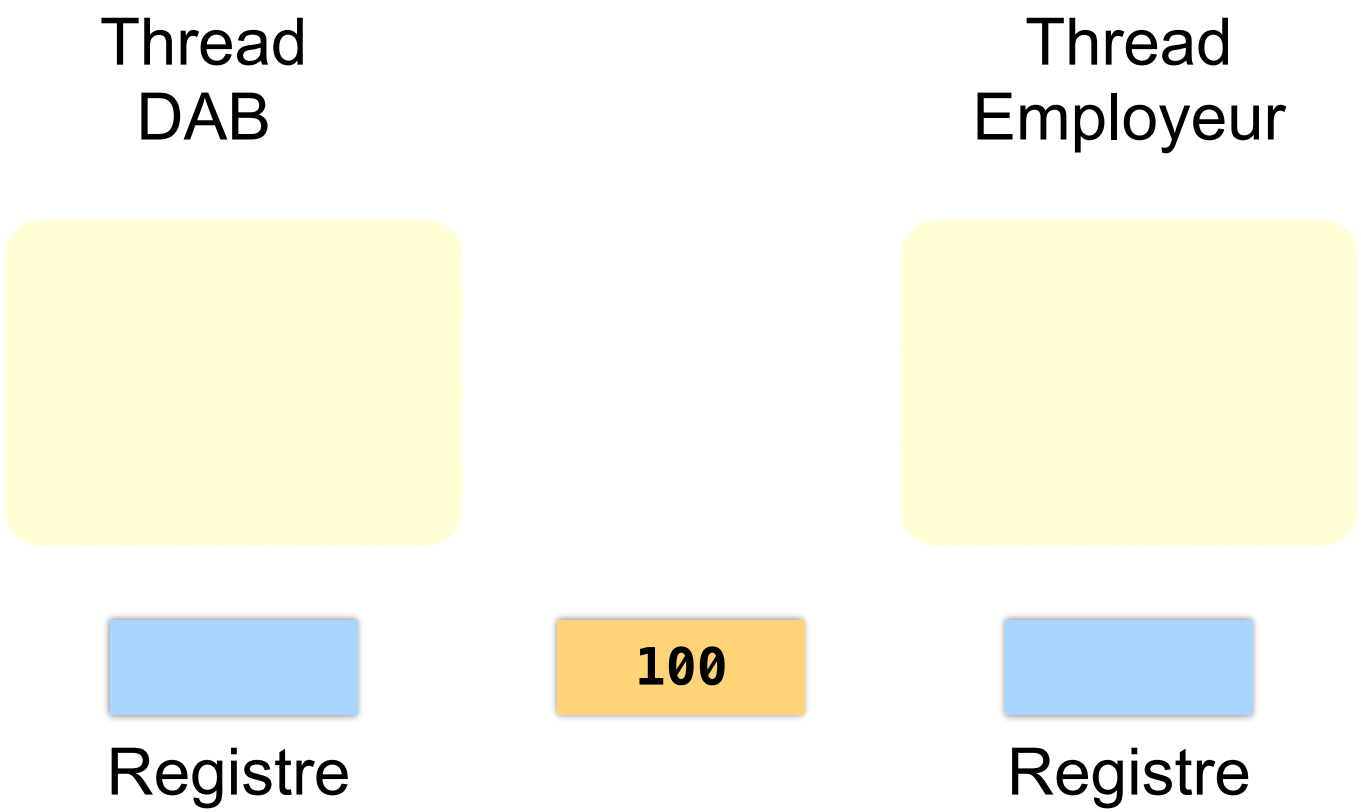
1. Chargement de l'attribut `solde` (en mémoire) dans un registre du processeur
2. Ajout de `montant` au registre

Version incorrecte

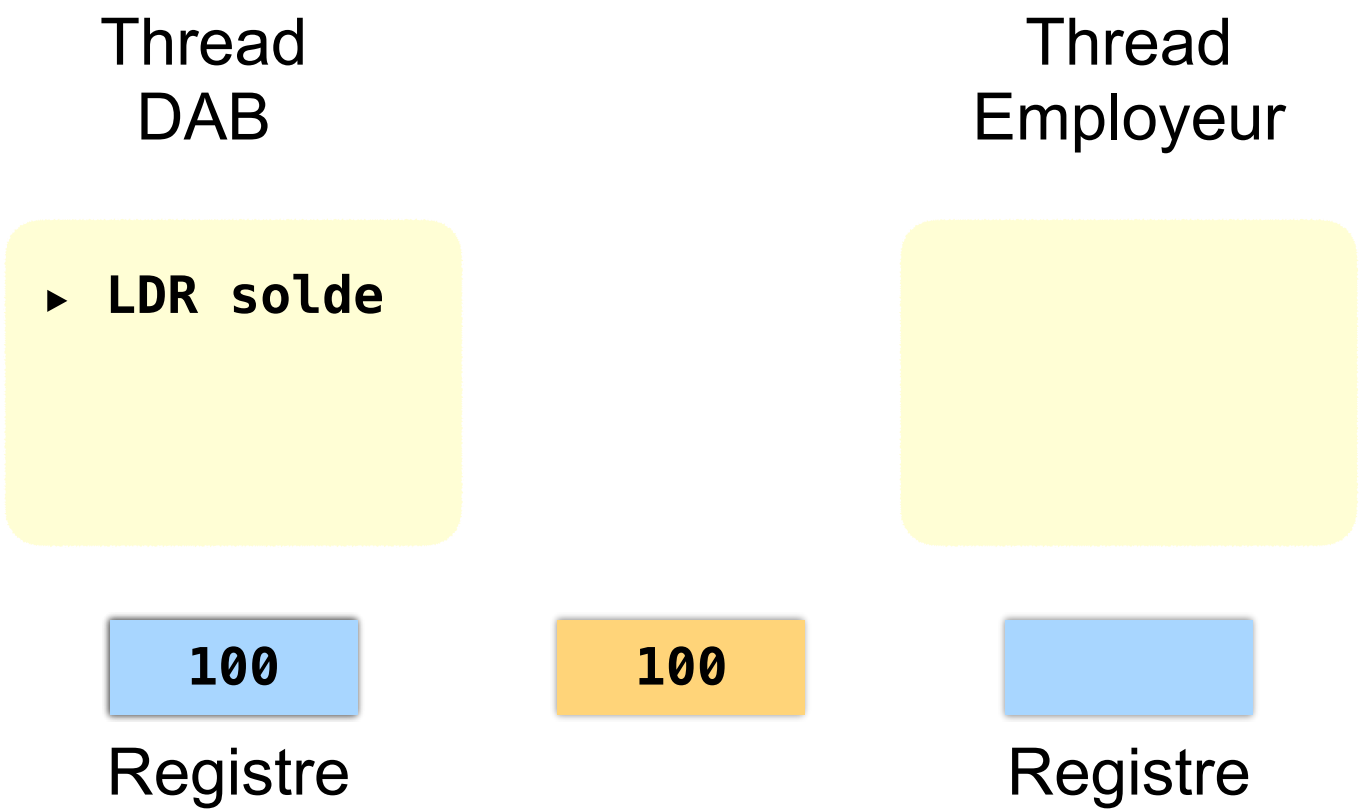
```
class Compte {
public:
  void credit( double montant ) {
    solde_ += montant;
  }

  void debit( double montant ) {
    solde_ -= montant;
  }

private:
  double solde_;
}
```

1. Chargement de l'attribut `solde` (en mémoire) dans un registre du processeur
2. Ajout de `montant` au registre
3. Transfert du contenu du registre en mémoire (dans l'attribut `solde`)

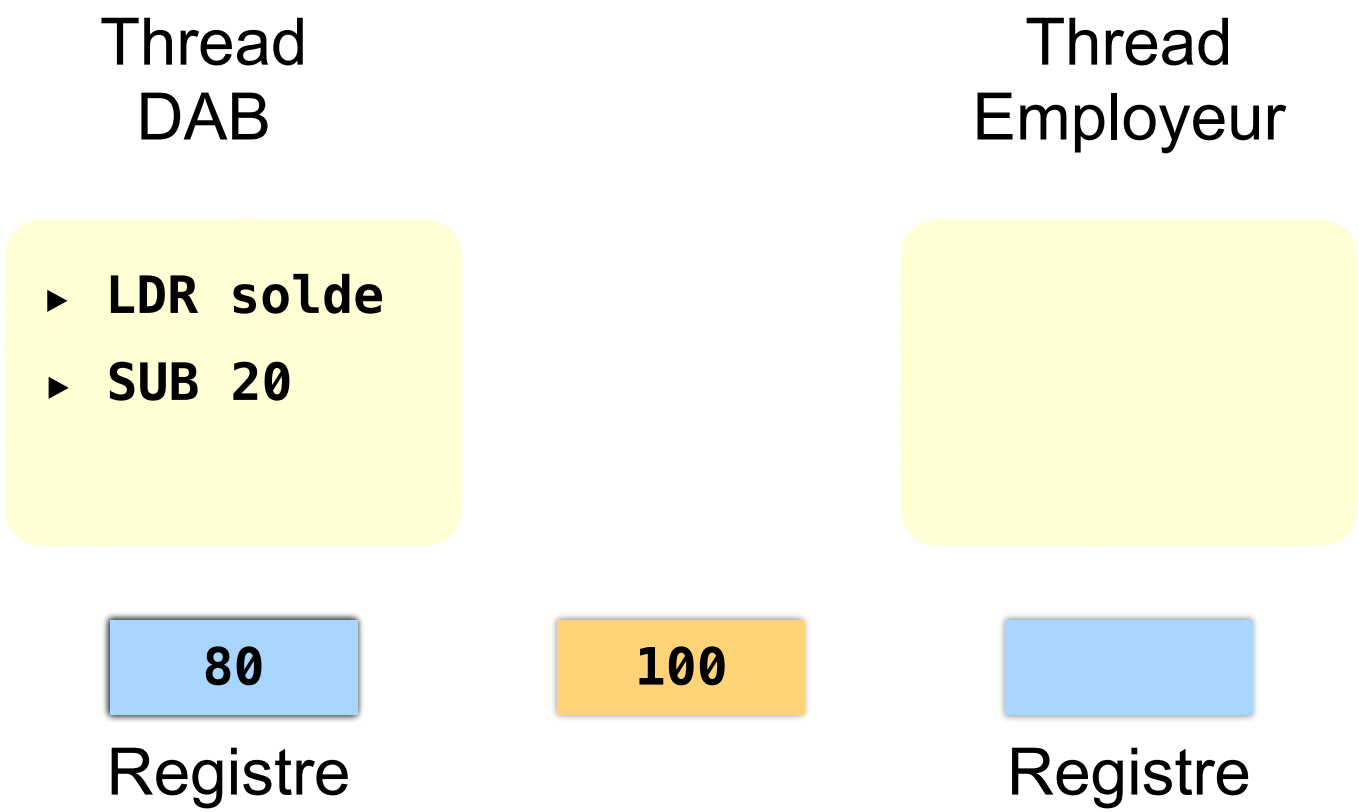
Exécution correcte



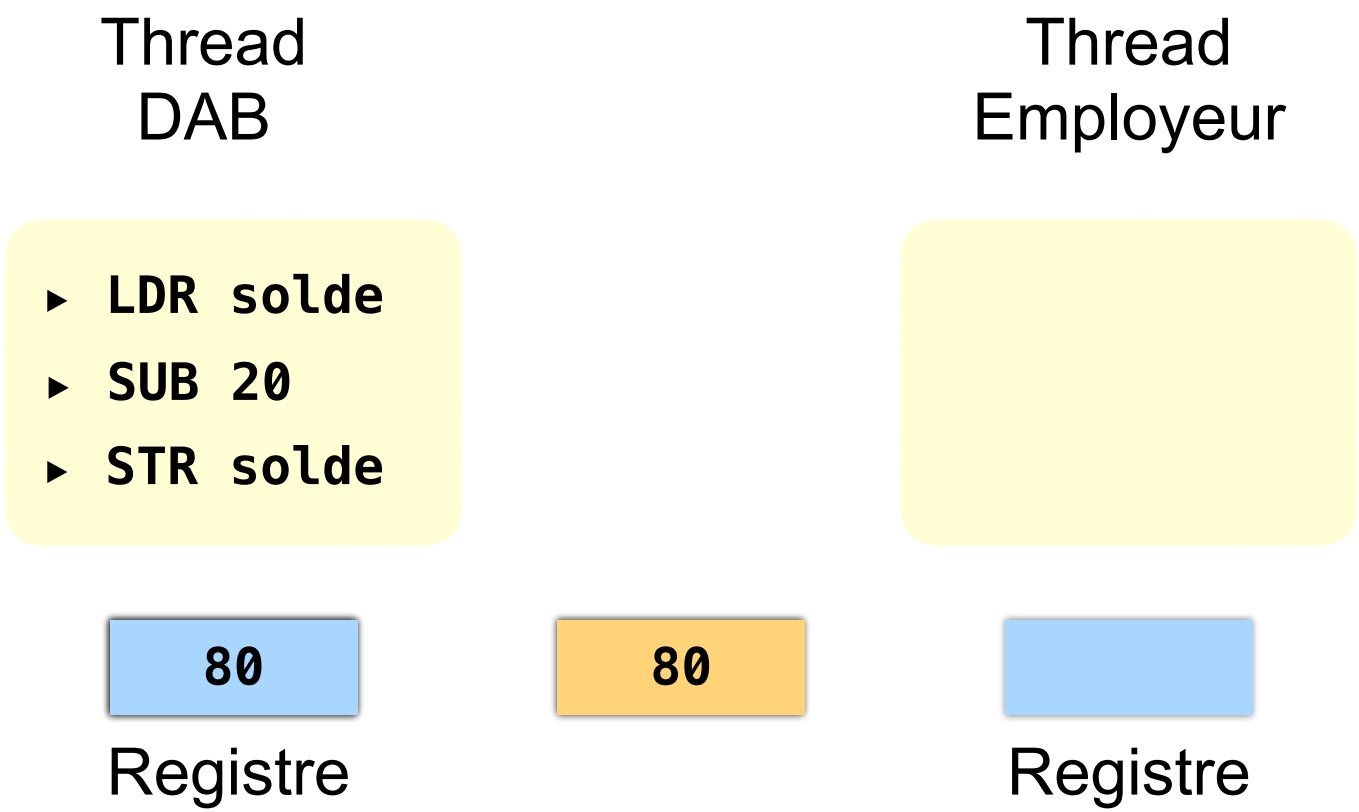
Exécution correcte



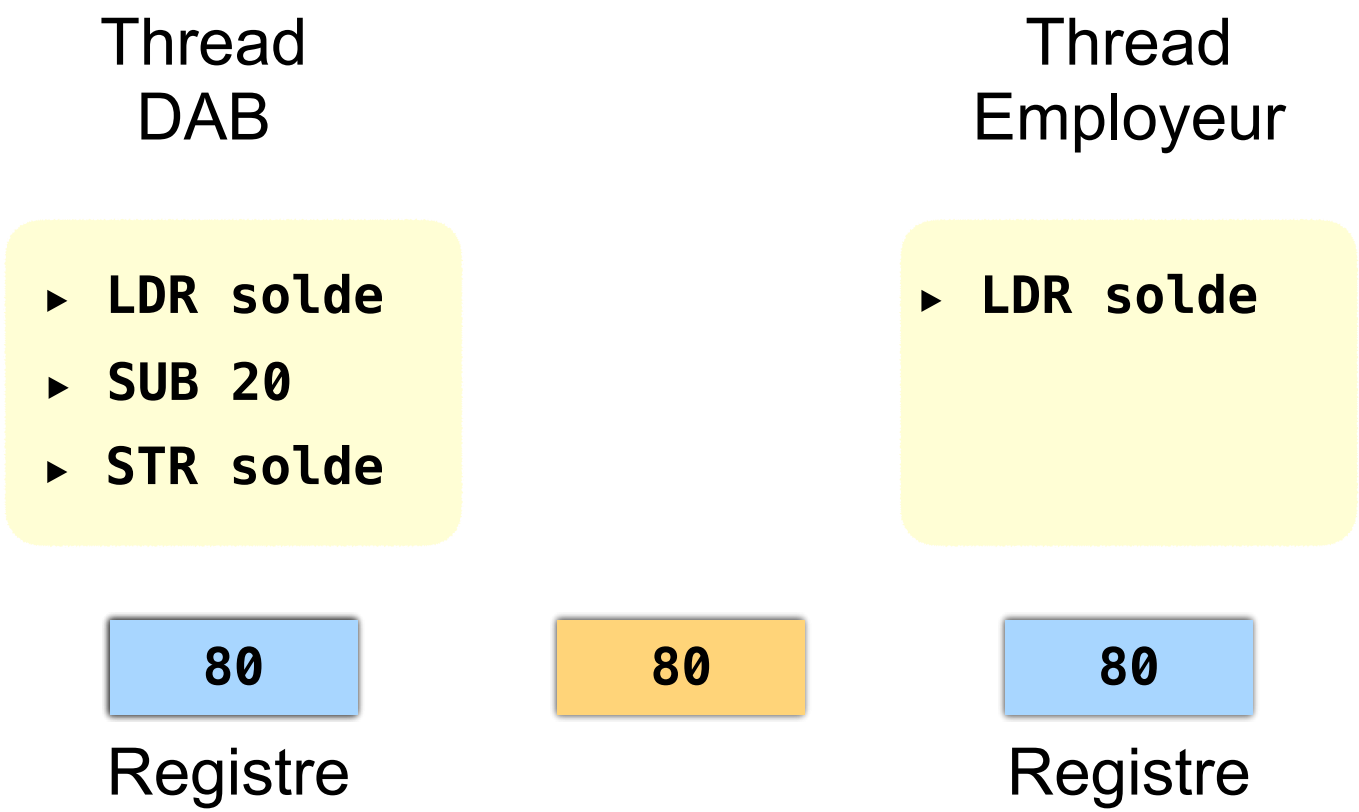
Exécution correcte



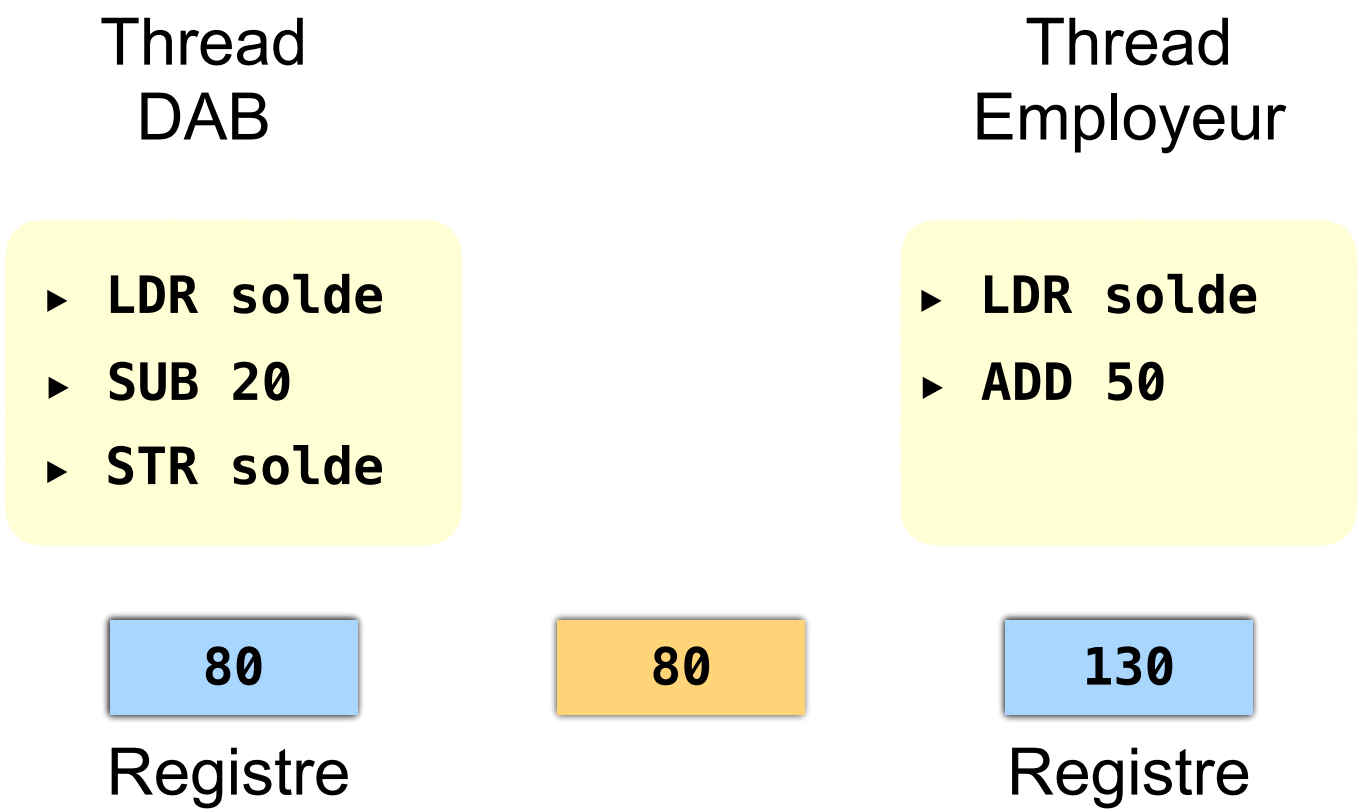
Exécution correcte



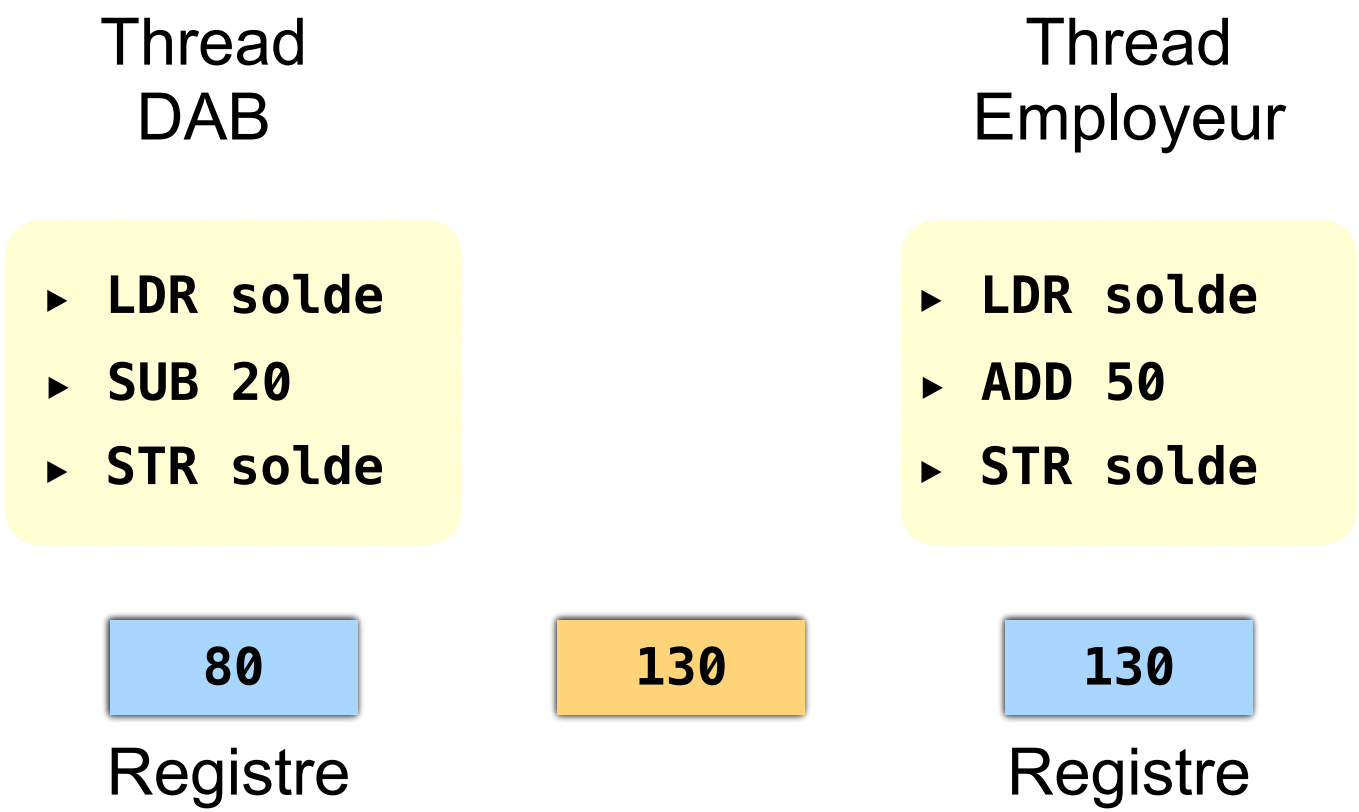
Exécution correcte



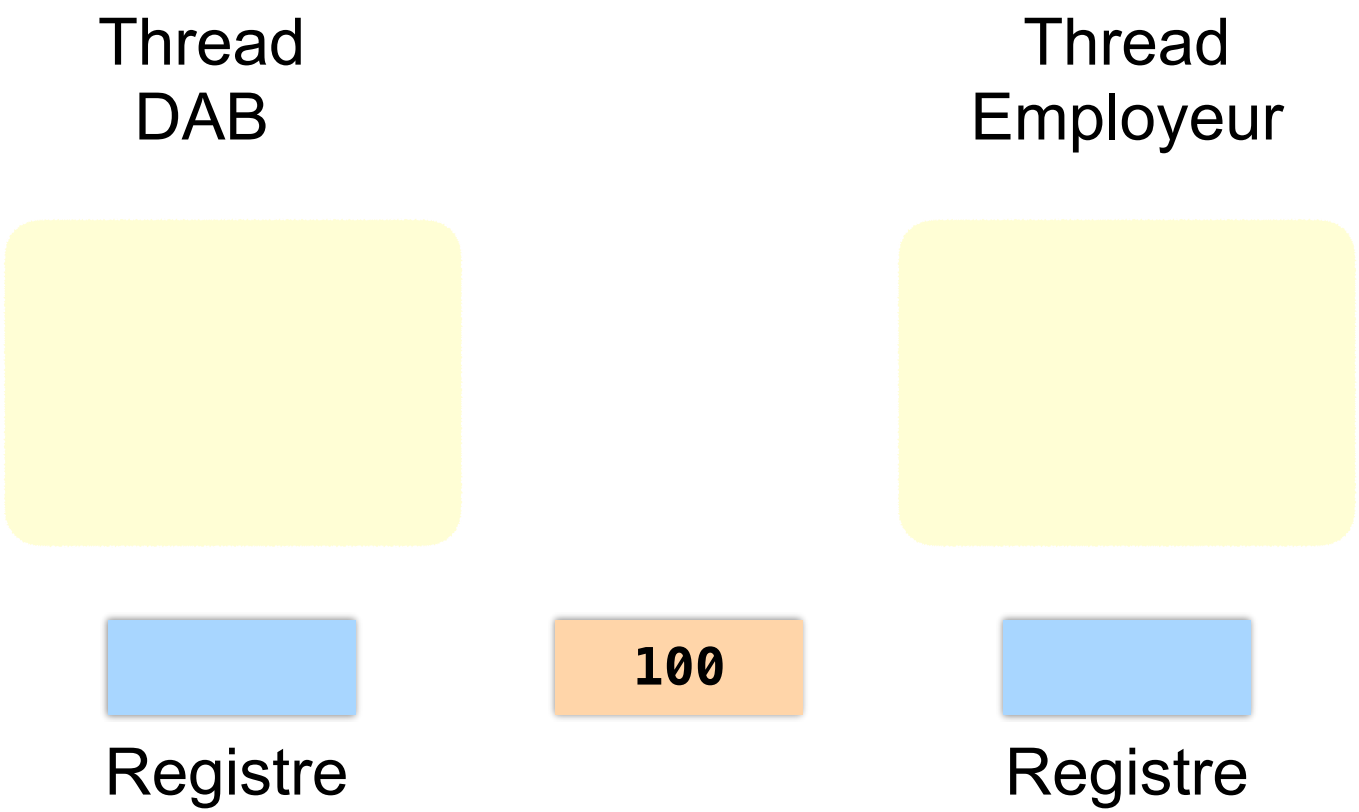
Exécution correcte



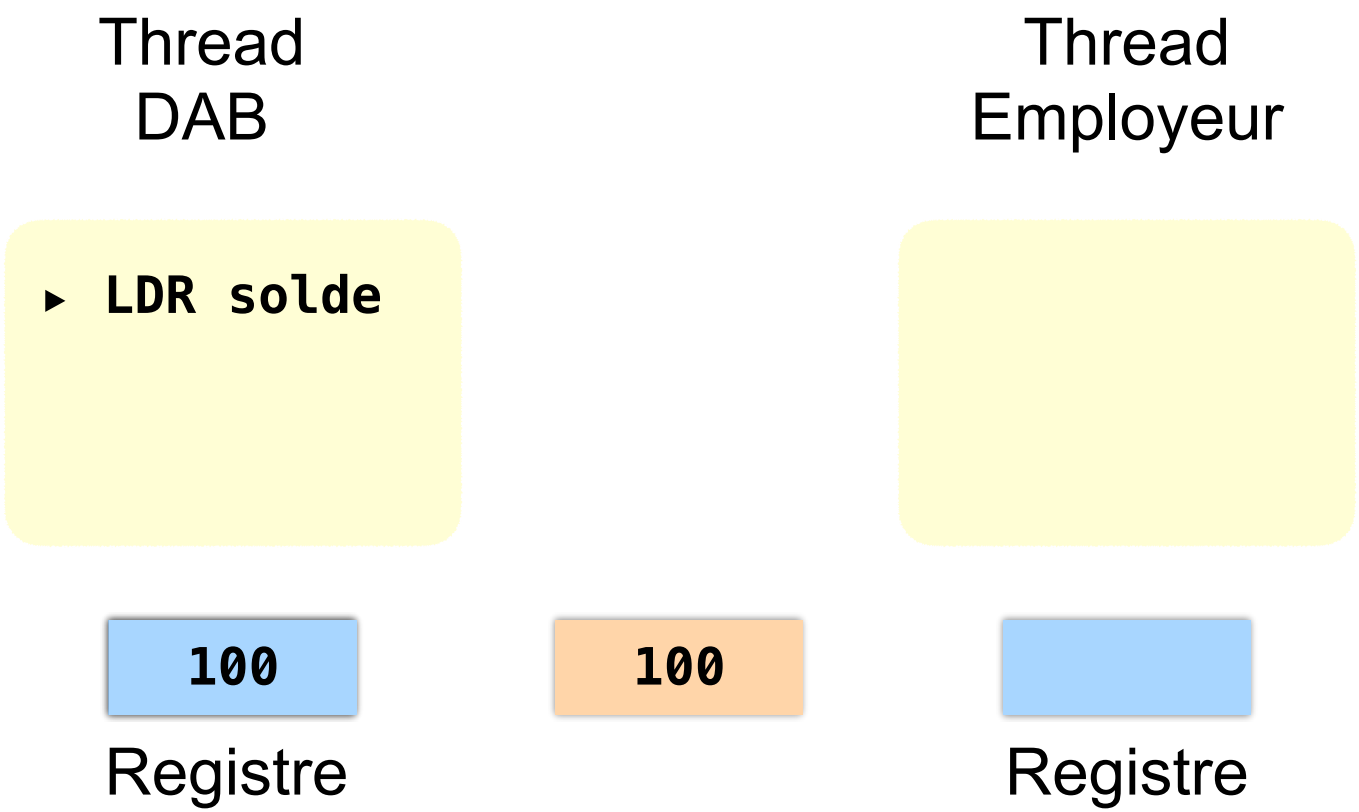
Exécution correcte



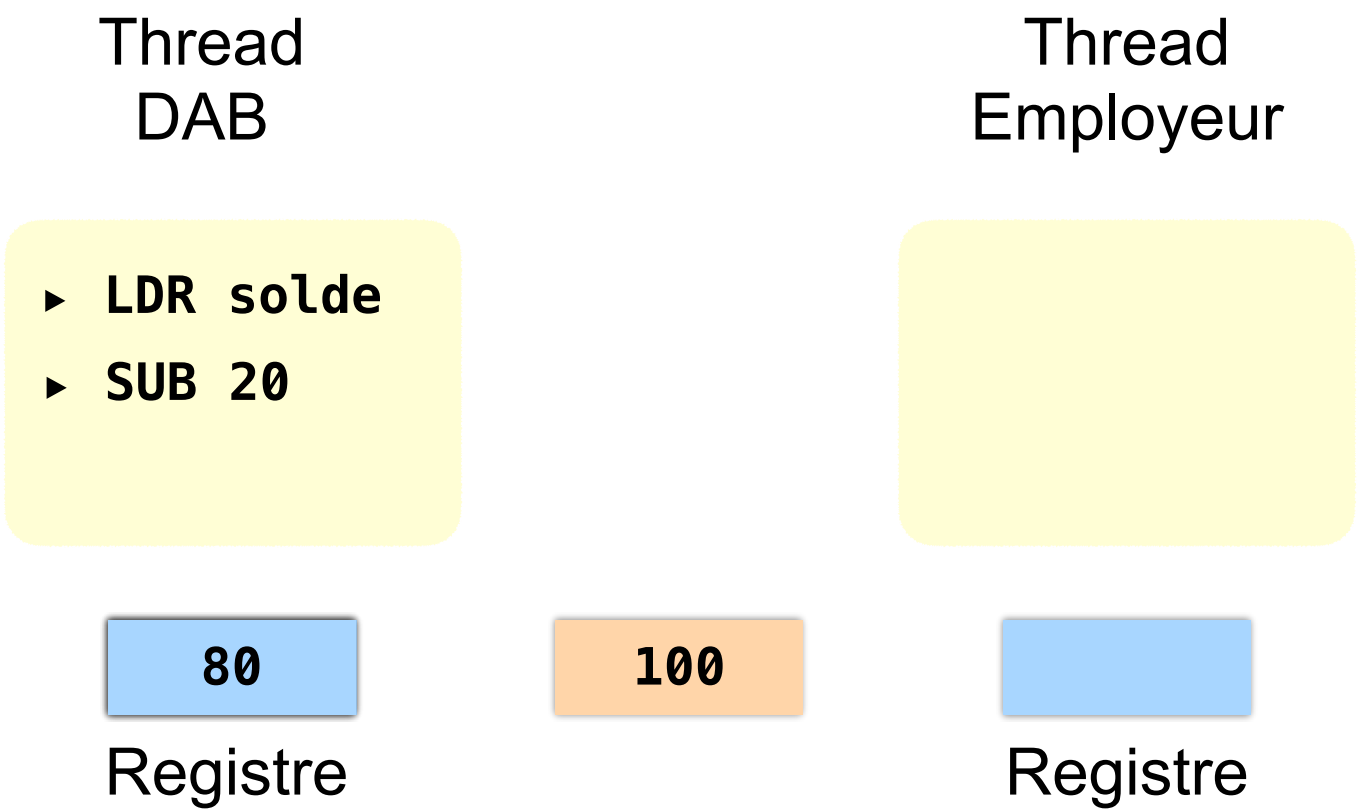
Exécution incorrecte



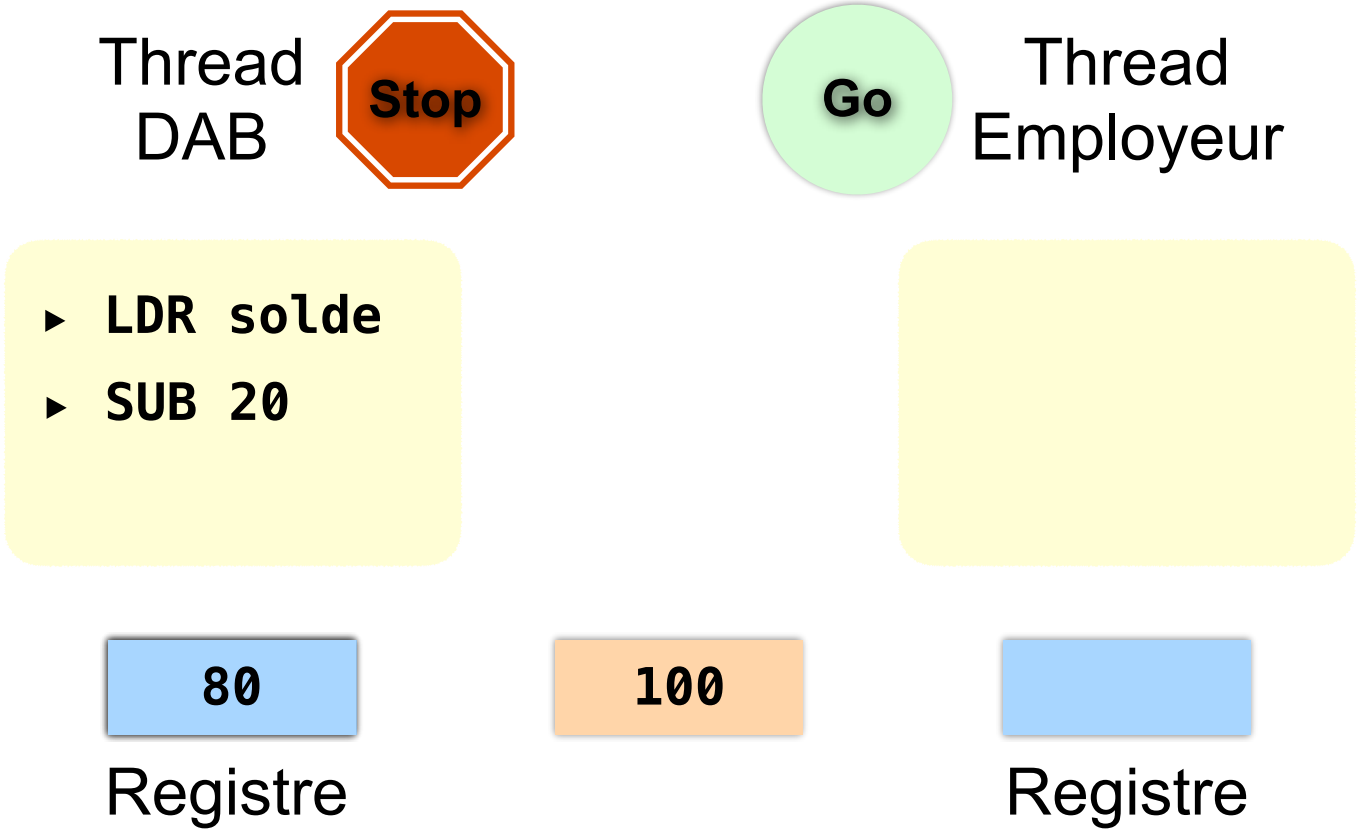
Exécution incorrecte



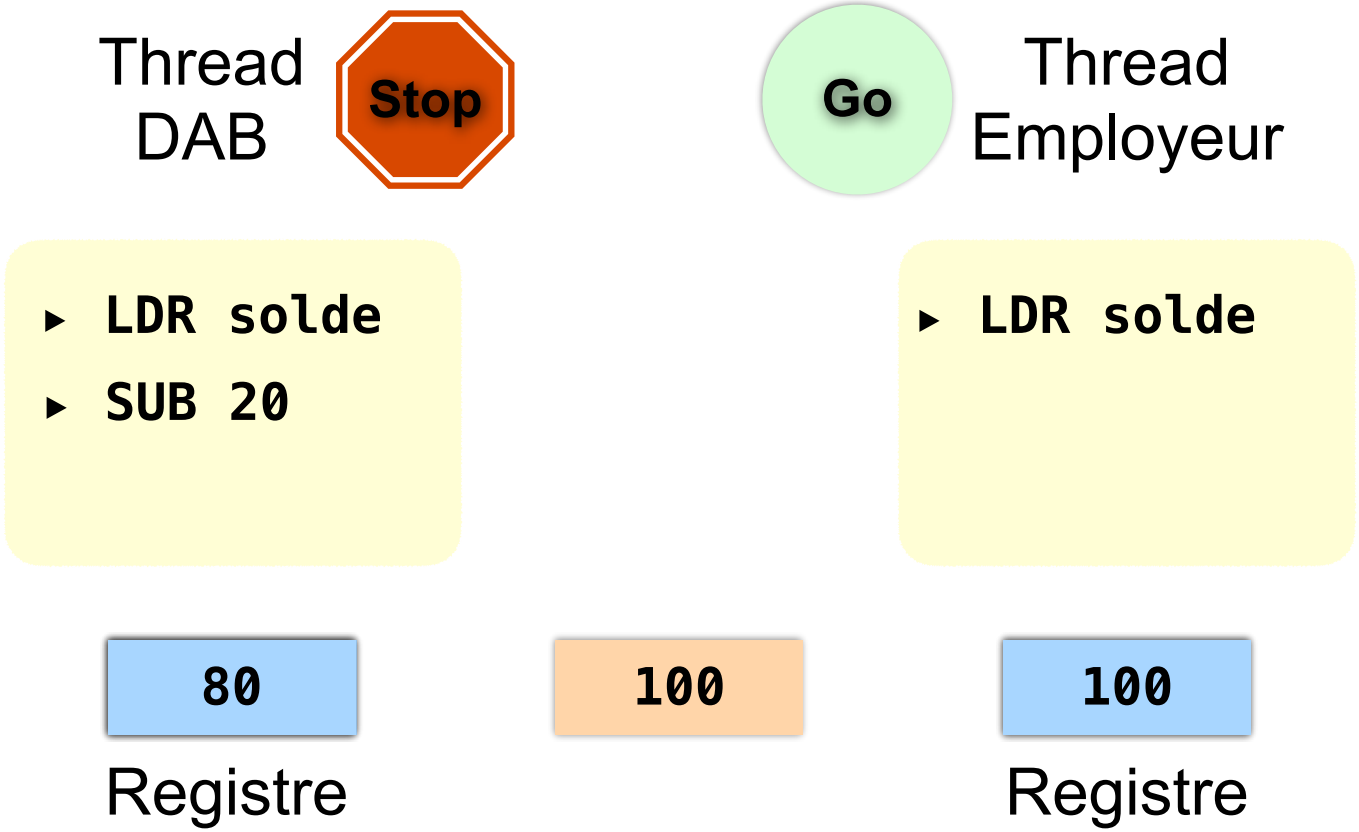
Exécution incorrecte



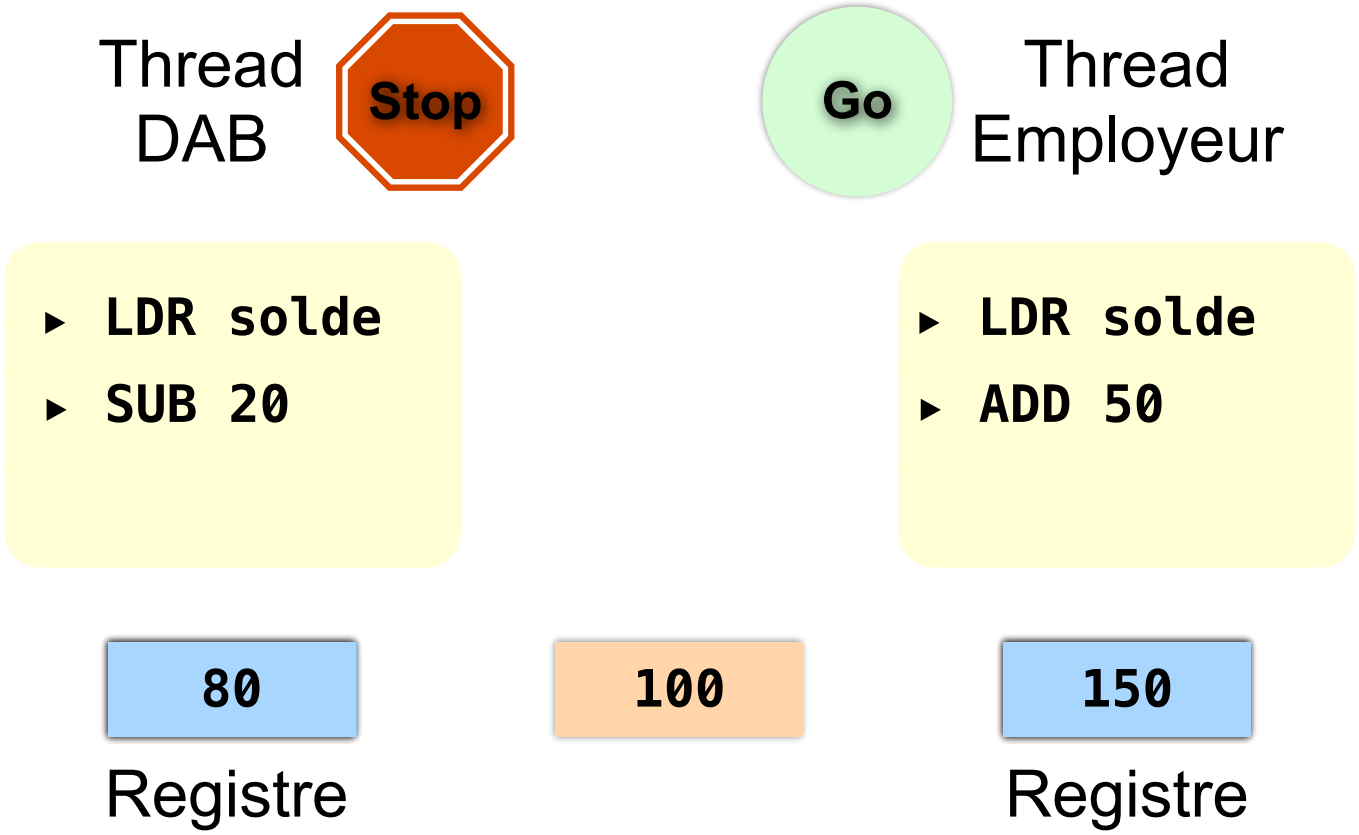
Exécution incorrecte



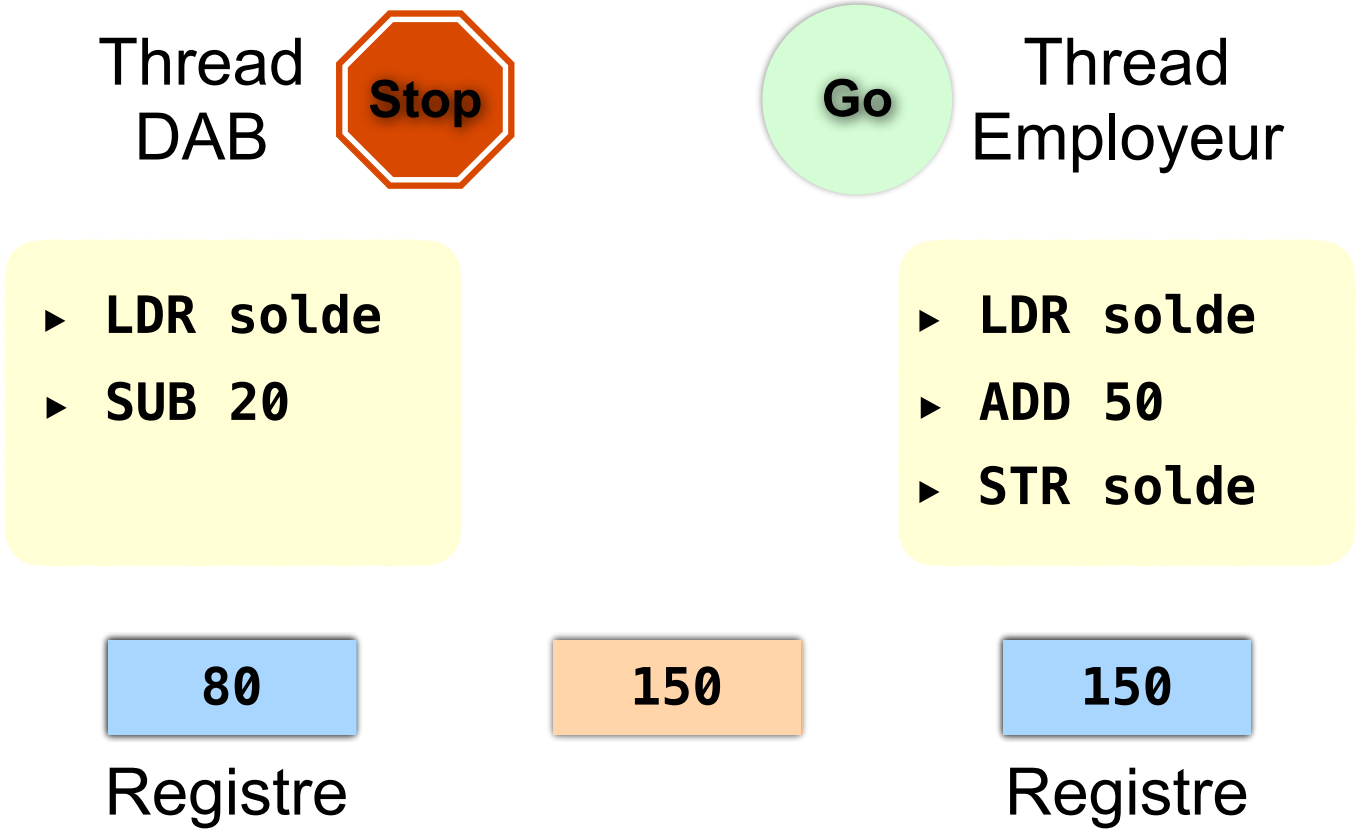
Exécution incorrecte



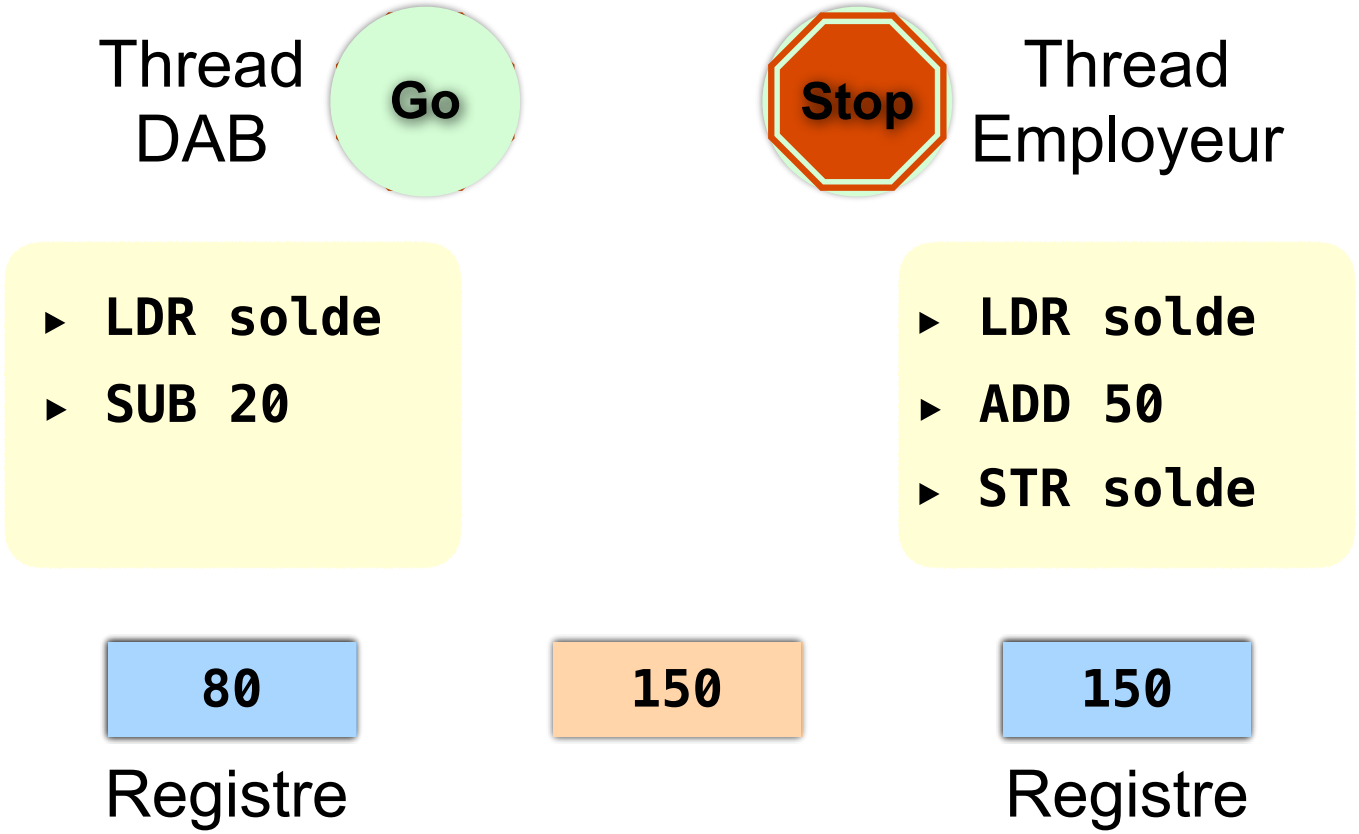
Exécution incorrecte



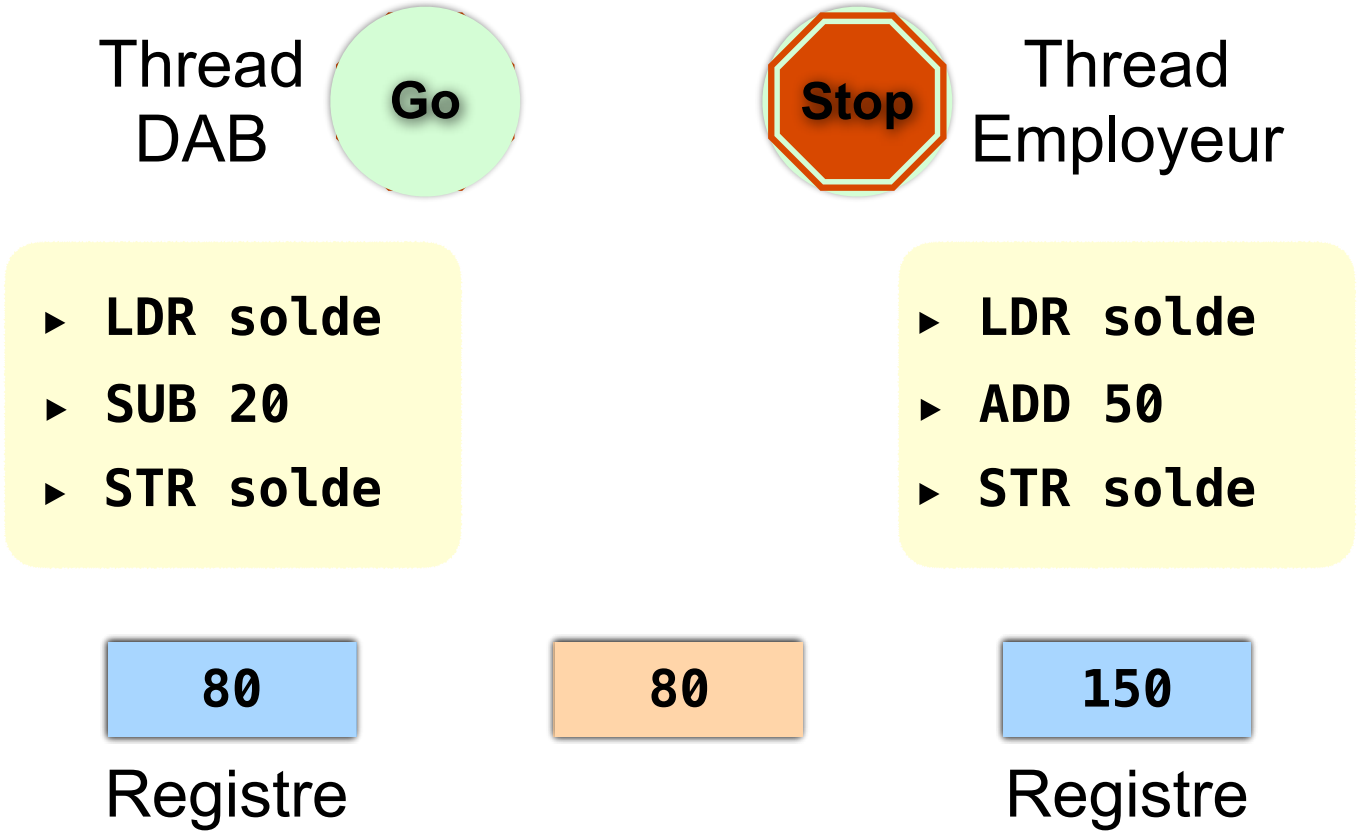
Exécution incorrecte



Exécution incorrecte



Exécution incorrecte



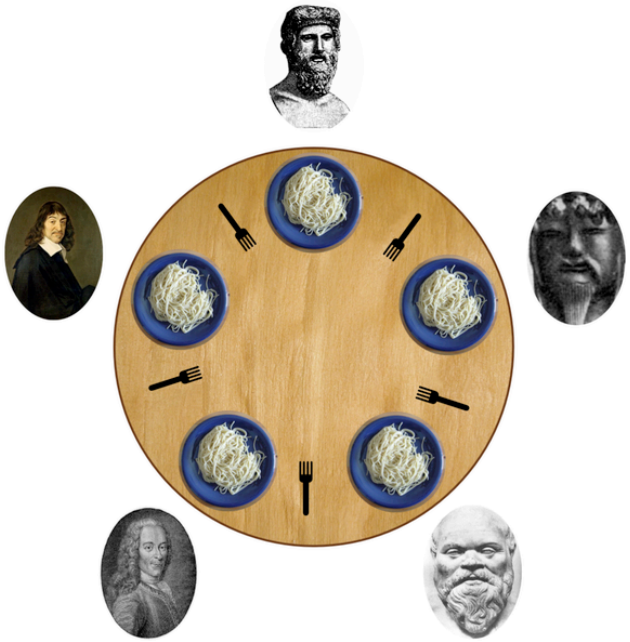
Démo

GitLab

`cours/progsyst/SharedVariable.cpp`

Le dîner des philosophes

- Interblocage
- Famine



Définitions

- Ressource critique
  - ressource qui ne doit pas être accédée par plusieurs processus ou threads simultanément
- Section critique
  - partie(s) de code accédant à une ressource critique
- Exclusion mutuelle
  - mécanisme permettant l'accès correct à une ressource critique

Propriétés

- Sûreté : au plus un seul processus utilise la ressource à un moment donné

- Vivacité :

  Un processus obtient l'accès immédiat à une ressource disponible

  Un processus obtient l'accès à la ressource au bout d'un temps fini

- Efficacité : un processus en attente d'une ressource n'utilise pas le processeur (il est bloqué)

Solutions

- Il existe des algorithmes (algorithme de Peterson, algorithme de Dekker), mais ceux-ci imposent une attente active (efficacité non respectée)

- C'est le système d'exploitation qui peut mettre un thread ou un processus à l'état bloqué, donc c'est lui qui doit fournir le service

- Une implémentation efficace impose l'existence d'une transaction de lecture/écriture au niveau du bus et l'existence d'une instruction correspondante sur le processeur (test-and-set, compare-and-swap…)

Plan

- Présentation
- Programmation sous Unix
  - Fichier
  - Processus
  - Communication
  - Threads
  - Synchronisation
    - Problématique
    - Synchronisation de threads
    - Synchronisation de processus
- Virtualisation et conteneurisation

Mutex POSIX

- Création

```
int pthread_mutex_init(
  pthread_mutex_t * mutex,
  const pthread_mutexattr_t * mutexattr
);
```

- Destruction du mutex

```
int pthread_mutex_destroy(
  pthread_mutex_t * mutex
);
```

- Acquisition du mutex

```
int pthread_mutex_lock(
  pthread_mutex_t * mutex
);
```

- Libération du mutex

```
int pthread_mutex_unlock(
  pthread_mutex_t * mutex
);
```

Compte avec mutex POSIX

```
class Compte {
public:
   Compte() { pthread_mutex_init( &m_, NULL ); }
  ~Compte() { pthread_mutex_destroy( &m_ ); }

  void credit( double montant ) {
    pthread_mutex_lock( &m_ );
    solde_ += montant;
    pthread_mutex_unlock( &m_ );
  }

  void debit( double montant ) {
    pthread_mutex_lock( &m_ );
    solde_ -= montant;
    pthread_mutex_unlock( &m_ );
  }

private:
  double solde_;
  pthread_mutex_t m_;
}
```

C++ : std::mutex

```
class mutex {
public:
  void lock();
  bool try_lock();
  void unlock();
  // ...
};
class recursive_mutex { /* ... */ };

class timed_mutex {
public:
  bool try_lock_for( /* ... */ );
  bool try_lock_until( /* ... */ );
  // ...
};
class recursive_timed_mutex { /* ... */ };
```

C++ : std::lock_guard

```
template< typename Mutex >
class lock_guard {
public:
  explicit lock_guard( Mutex & m );
  ~lock_guard();

  lock_guard( const lock_guard & ) = delete;
  void operator=( const lock_guard & ) = delete;
};
```

Compte avec mutex C++

```
class Compte {
public:
  void credit( double montant ) {
    // Ne pas faire
    // std::lock_guard{ m_ };
    std::lock_guard lock{ m_ };
    solde_ += montant;
  }

  void debit( double montant ) {
    std::lock_guard lock{ m_ };
    solde_ -= montant;
  }

private:
  double solde_;
  std::mutex m_;
}
```

Producteur - Consommateur

- Le producteur dépose des données dans une boîte à lettres de taille fixe.
- Le consommateur les retire.
- Synchronisation :
  - le consommateur doit attendre qu'une donnée soit présente dans la boîte à lettres ;
  - le producteur doit attendre d'avoir de la place dans la boîte à lettres.
- Variantes : plusieurs producteurs et consommateurs.

Moniteur

- Hansen (1973), Hoare (1974)

- Un moniteur est un module constitué de :
  - fonctions d'accès à la ressource critique en exclusion mutuelle (un verrou)
  - condition booléenne et services permettant le blocage d'un processus/thread (avec libération du verrou) si la condition n'est pas remplie et réactivation par un autre processus/thread quand la condition peut être remplie (l'accès à la ressource critique reste soumise au verrou)

- Ce mécanisme doit être intégré au langage
- C'est par exemple le mécanisme de base de synchronisation en Java (classe `Object`, mot clef `synchronized`).

Variable de condition POSIX

- Création

```
int pthread_cond_init(
  pthread_cond_t * cond,
  pthread_condattr_t * attr
);
```

- Destruction

```
int pthread_cond_destroy(
  pthread_cond_t * cond
);
```

- Attente

```
int pthread_cond_wait(
  pthread_cond_t * cond,
  pthread_mutex_t * mutex
);
```

- Signalisation

```
int pthread_cond_signal(
  pthread_cond_t * cond
);
```

Exemple POSIX (1)

```
pthread_cond_t  data_place_cond;
pthread_mutex_t data_place_mutex;
volatile int data_available = 0;

int main()
{
  pthread_t cons, prod;

  pthread_cond_init( &data_place_cond, NULL );
  pthread_mutex_init( &data_place_mutex, NULL );

  pthread_create( &cons, NULL, &consumer, NULL );
  pthread_create( &prod, NULL, &producer, NULL );
  pthread_join( cons, NULL );
  pthread_join( prod, NULL );
}
```

Exemple POSIX (2)

```
void * producer( void * producer_thread_data ) {
  while( !done() ) {
    create_data();

    pthread_mutex_lock( &data_place_mutex );
    if( data_available == 1 ) {
      pthread_cond_wait(
        &data_place_cond,
        &data_place_mutex );
    }
    put_data_into_place();
    data_available = 1;
    pthread_cond_signal( &data_place_cond );
    pthread_mutex_unlock( &data_place_mutex );
  }
}
```

Exemple POSIX (3)

```
void * consumer( void * consumer_thread_data ) {
  while( !done() ) {
    pthread_mutex_lock( &data_place_mutex );
    if( data_available == 0 ) {
      pthread_cond_wait(
        &data_place_cond,
        &data_place_mutex );
    }
    get_data_from_place();
    data_available = 0;
    pthread_cond_signal( &data_place_cond );
    pthread_mutex_unlock( &data_place_mutex );

    process_data();
  }
}
```

C++ : std::condition_variable

```
class condition_variable {
public:
  void notify_one();
  void notify_all();

  void wait( unique_lock< mutex > & lock );
  bool wait_until( /* ... */ );
  bool wait_for( /* ... */ );
  // ...
};

class condition_variable_any { /* ... */ };
```

C++ : std::unique_lock

```
template< typename Mutex >
class unique_lock {
public:
  void lock();
  bool try_lock();
  void unlock();

  bool try_lock_for( /* ... */ );
  bool try_lock_until( /* ... */ );
  // ...
};
```

C++ : exemple (1)

```
std::condition_variable data_place_cond;
std::mutex              data_place_mutex;
volatile bool data_available{ false };

int main()
{
  std::jthread cons{ consumer{} }, prod{ producer{} };
}
```

C++ : exemple (2)

```
struct producer {
  void operator()() {
    while( !done() ) {
      create_data();
      {
        std::unique_lock lock{ data_place_mutex };
        if( data_available ) {
          cond_place_empty.wait( lock );
        }
        put_data_into_place();
        data_available = true;
        cond_place_full.notify_one();
      }
    }
  }
};
```

C++ : exemple (3)

```
struct consumer {
  void operator()() {
    while( !done() ) {
      {
        std::unique_lock lock{ data_place_mutex };
        if( ! data_available ) {
          cond_place_full.wait( lock );
        }
        get_data_from_place();
        data_available = false;
        cond_place_empty.notify_one();
      }
      process_data();
    }
  }
};
```

Démo

GitLab

`cours/progsyst/ProdCons.cpp`

Variable partagée (KO)

```
void f( int & x ) { /*...*/ x += 3; }

int main()
{
  int n{ 2 };
  std::jthread t1{ f, std::ref( n )};
  std::jthread t2{ f, std::ref( n )};
}
```

Variable partagée (OK)

```
class ProtectedInt {
public:
  ProtectedInt( int n ) : n_{ n } {}
  void operator+=( int i ) {
    std::lock_guard< std::mutex > lock( mutex_ );
    n_ += i;
  }
private:
  int n_;      std::mutex mutex_;
};

void f( ProtectedInt & x ) { /*...*/ x += 3; }

int main()
{
  ProtectedInt n{ 2 };
  std::jthread t1{ f, std::ref( n )}, t2{ f, std::ref( n )};
}
```

std::atomic

```
void f( std::atomic_int & x ) {
  /*...*/
  x += 3;
}

int main()
{
  std::atomic_int n{ 2 };
  std::jthread t1{ f, std::ref( n )};
  std::jthread t2{ f, std::ref( n )};
}
```

Plan

- Présentation
- Programmation sous Unix
  - Fichier
  - Processus
  - Communication
  - Threads
  - Synchronisation
    - Problématique
    - Synchronisation de threads
    - Synchronisation de processus
- Virtualisation et conteneurisation

Différent des threads ?

- Les threads d'un processus partagent le même espace mémoire
  - Une donnée (par exemple un mutex) à la même adresse dans tous les threads

- Certains mécanismes de communication intègrent un mécanisme de synchronisation
  - pipe
  - socket

- Pour la synchronisation entre processus, deux approches sont possibles :
  - Mettre les objets de synchronisation dans un espace mémoire partagé entre processus
  - Identifier (nommer) l'objet de synchronisation au niveau du système d'exploitation

Via un espace mémoire partagé

- L'objet de synchronisation ne doit pas mémoriser d'adresse
  - Exemple : pas de fonctions virtuelles C++
- Sur Linux, un attribut lors de la création de l'objet de synchronisation peut être utilisé :

```
int pthread_mutexattr_setpshared(
  pthread_mutexattr_t * attr,
  int pshared
);
```

- Pour les variables de conditions, POSIX le prévoit :

```
int pthread_condattr_t_setpshared(
  pthread_condattr_t *attr,
  int pshared
);
```

Mais ce n'est pas implémenté sous Linux

Via un nom

- Il n'y a pas de mutex ou de variable de condition nommés sur Linux
- Pour les mutex, il est possible de contourner le problème en utilisant un fichier et un appel système permettant de bloquer l'accès à ce fichier par un autre processus :

```
int fcntl(
  int fd,
  int op,
  ...
);
```

- Pour les variables de condition, c'est un peu plus compliqué...
- En C++, des bibliothèques offrent des solutions multiplateformes :
  - `boost::interprocess::named_mutex`
  - `boost::interprocess::named_condition`

Sémaphore

- Un autre mécanisme de synchronisation est couramment utilisé : le sémaphore (Djikstra, 1965)
- Un sémaphore est constitué :
  - D'un compteur (qui représente le nombre de ressources disponibles s'il est positif ou demandées s'il est négatif)
  - D'une file d'attente de processus (de blocs de contexte)
  - De 3 opérations atomiques (non interruptibles)
    - `init( sem, val )` : initialise le compteur à `val` et la file à vide
    - `P( sem )` : décrémente le compteur ; si celui-ci est strictement négatif, bloque le processus et le met dans la file d'attente
    - `V( sem ) :` incrémente le compteur ; si celui-ci est négatif ou nul, libère un processus de la file d'attente
- Un mutex peut être réalisé avec un sémaphore binaire (compteur initial à 1) sans la contrainte que c'est le même thread qui doit faire le `lock()` et le `unlock()`.

POSIX : sémaphore anonyme

- Les sémaphores anonymes sont accessibles uniquement entre processus père-fils et les threads
  - Création

```
int sem_init(
  sem_t * sem,
  // pshared == 0 ? threads : processus
  int pshared,
  unsigned int value
);
```

  - Destruction

```
int sem_destroy( sem_t * sem );
```

POSIX : sémaphore nommé

- Les sémaphores nommés sont implémentés par un fichier et sont accessibles par tout les processus/threads
  - Création

```
sem_t * sem_open(
  char * name,
  int oflag,
  mode_t mode,
  int val
);
```

  - Destruction

```
int sem_destroy( sem_t * sem );
```

POSIX : opérations sur sémaphores

- Ces deux types de sémaphores offrent les services suivants :
  - Demande d'un jeton

```
int sem_wait( sem_t * sem );
```

  - Libération d'un jeton

```
int sem_post( sem_t * sem );
```

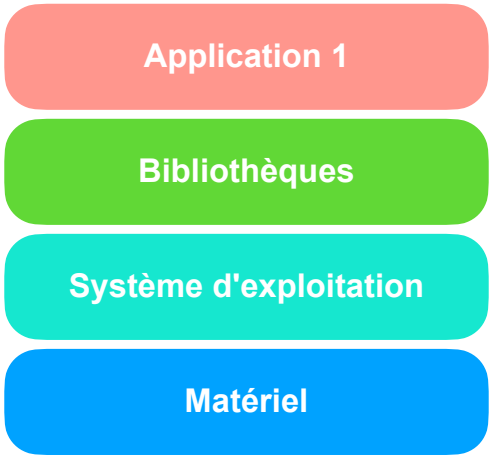
  - Nombres de jetons disponibles

```
int sem_getvalue( sem_t * sem, int * sval );
```

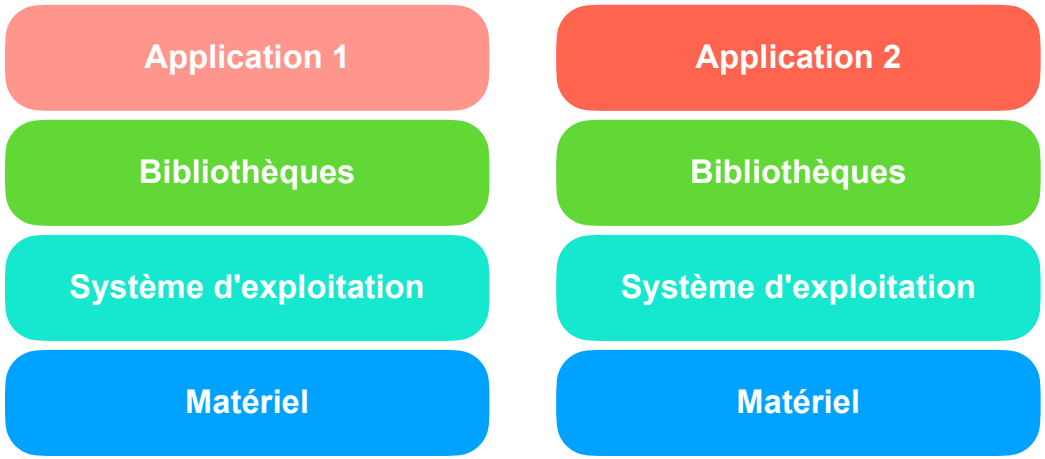
Plan

- Présentation
  - Introduction
  - Historique
- Programmation sous Linux
  - Fichier
  - Processus
  - Communication
  - Threads
  - Synchronisation
- Virtualisation et conteneurisation

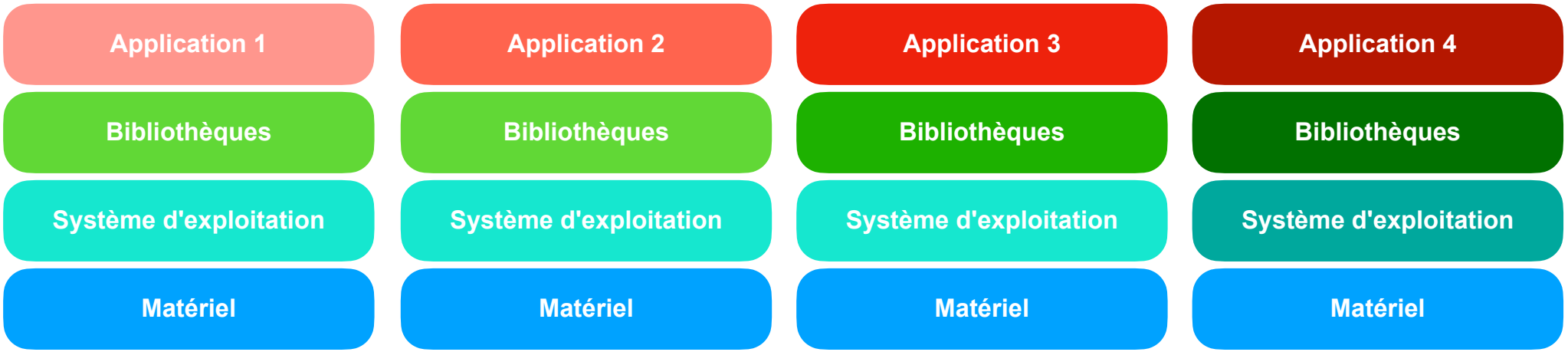
Système(s) unique(s)



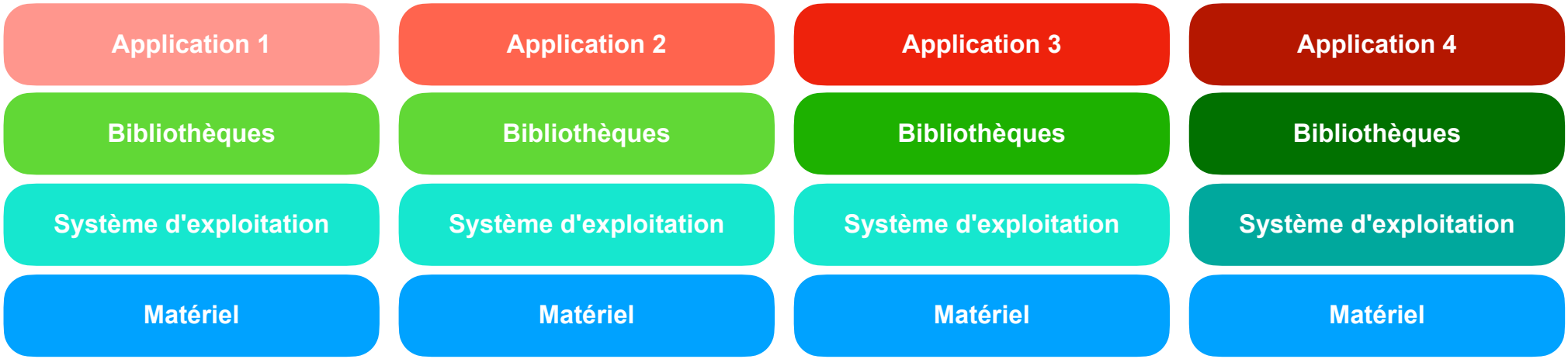
Système(s) unique(s)



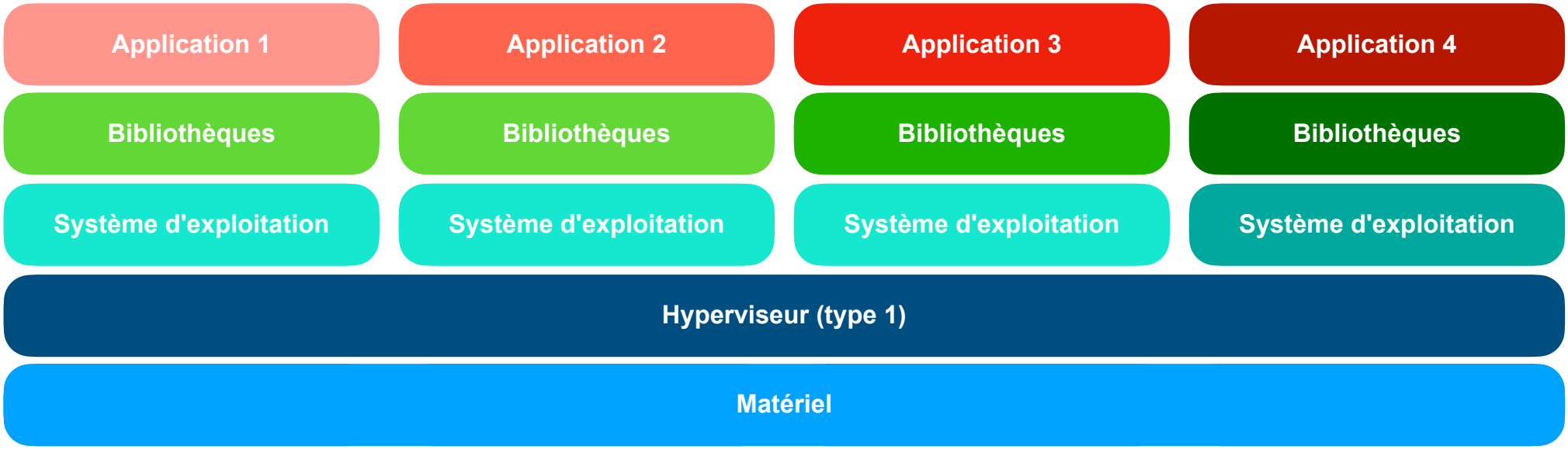
Système(s) unique(s)



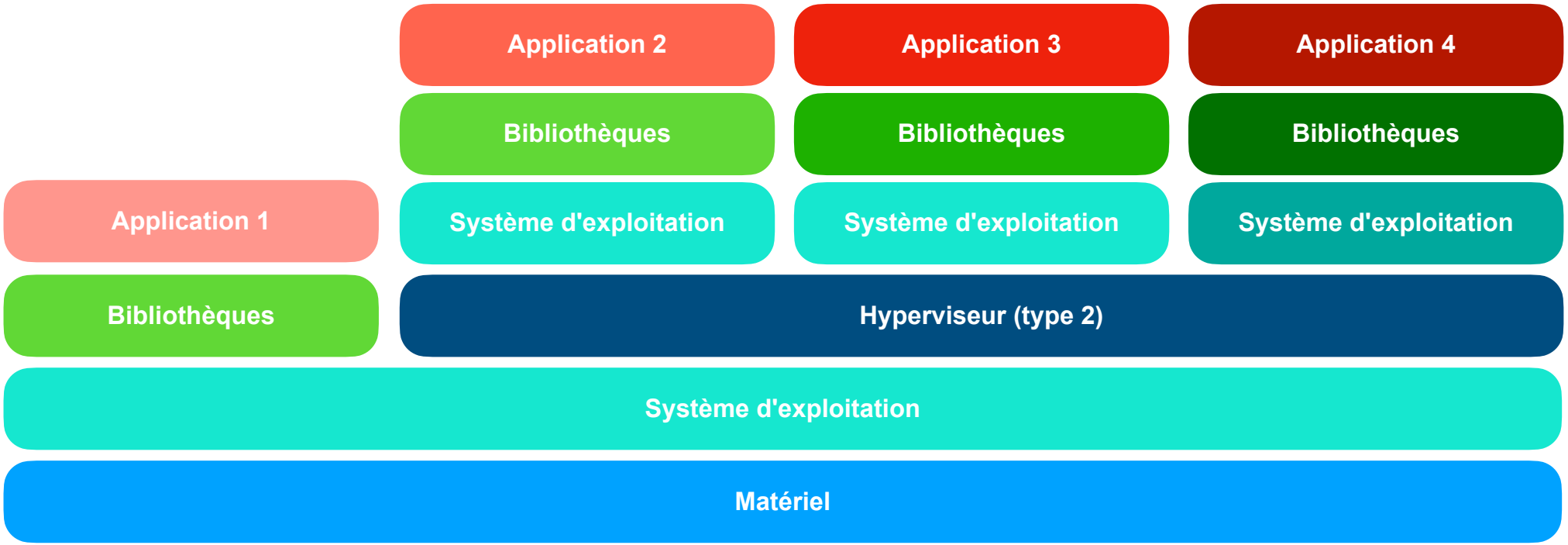
Hyperviseur (type 1)



Hyperviseur (type 1)



Hyperviseur (type 2)

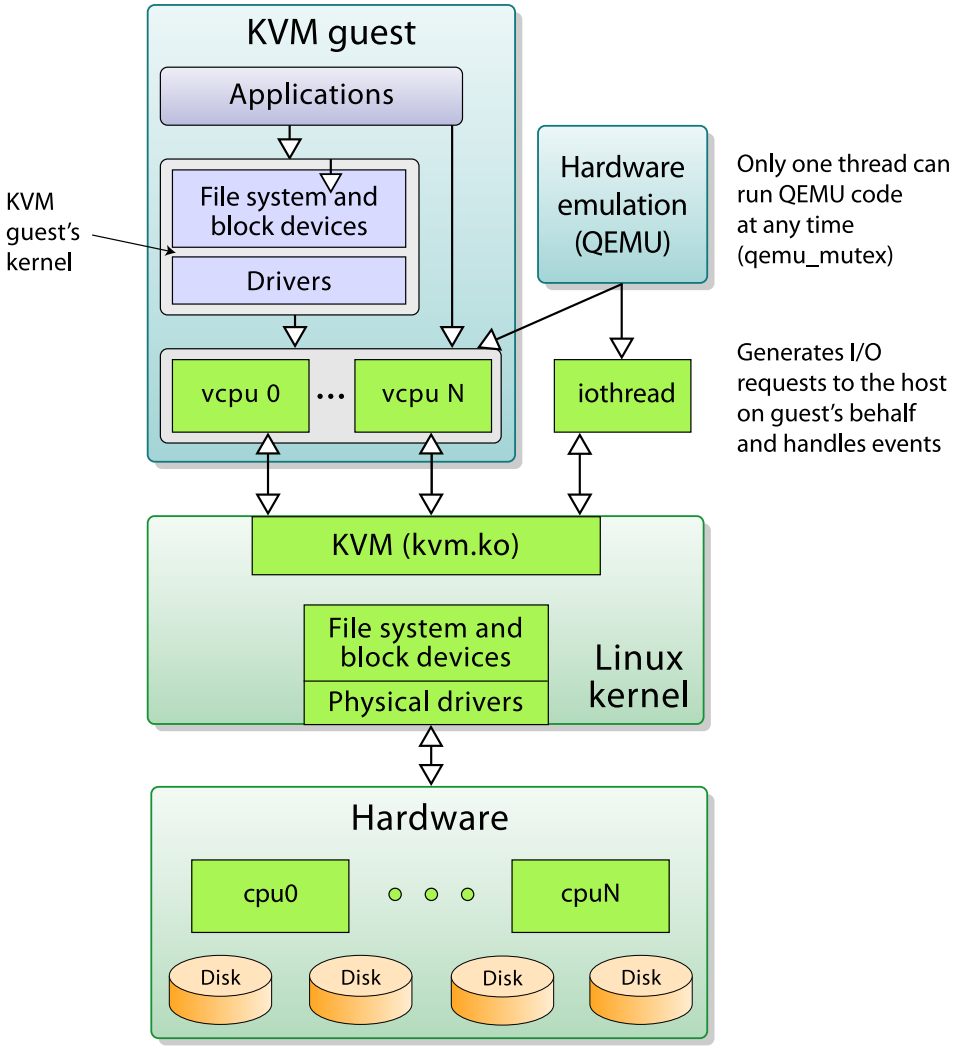


Virtualisation

- Système hôte et système invité
- Virtualisation logicielle
  - Le système invité s'exécute dans l'espace utilisateur
    - Quand il utilise des instructions réservées au mode noyau, le système hôte les remplace par des simulations équivalentes
  - Paravirtualisation
    - Le système invité est modifié pour remplacer l'exécution des instructions réservées au mode noyau par des appels au système hôte
    - Virtio : standard pour offrir au système invité les accès aux composants périphériques (disque, cartes réseau…)
- Virtualisation matérielle
  - Instructions spécifiques au niveau du processeur

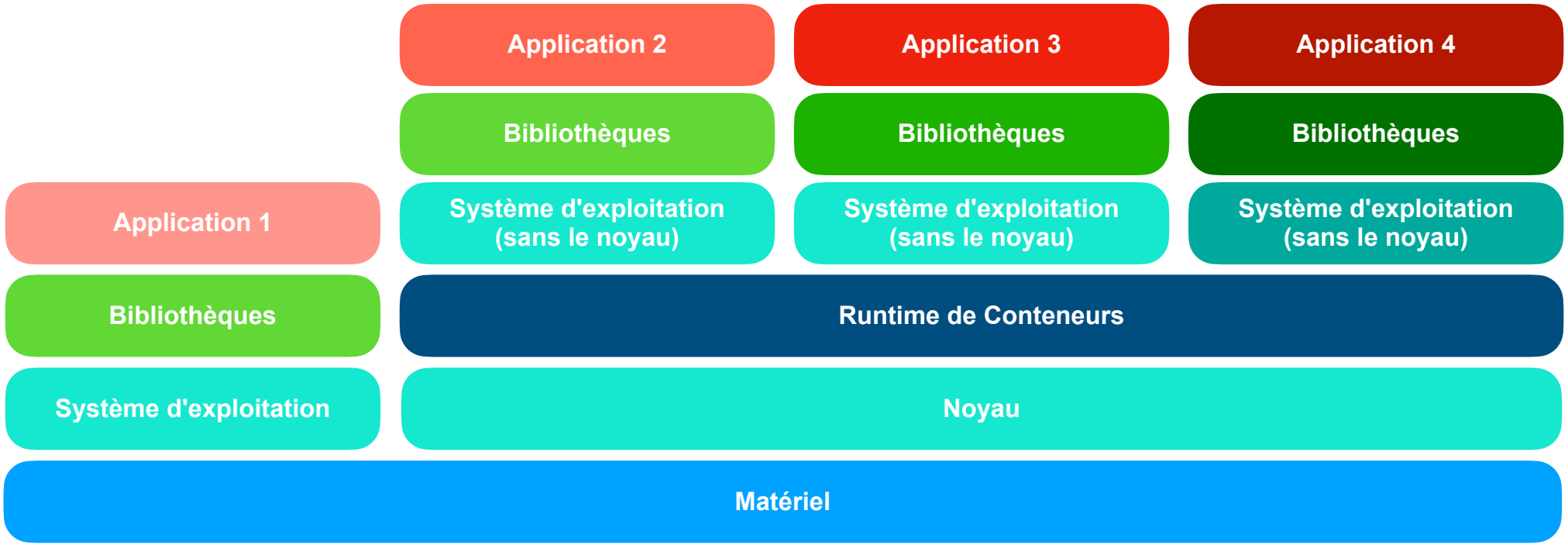
Exemple KVM/QEMU

- KVM (Kernel-based Virtual Machine)
  - Module du noyau Linux présent depuis février 2007
  - Permet au noyau Linux de fournir les services d'un hyperviseur type 1
- QEMU (Quick Emulator)
  - Émulation d'une machine complète (y compris avec un processeur différent)
  - S'exécute en espace utilisateur
- KVM/QEMU
  - QEMU simule structure de la machine (le BIOS, la RAM, les disques, le réseau), mais c'est KVM qui exécute nativement les instructions

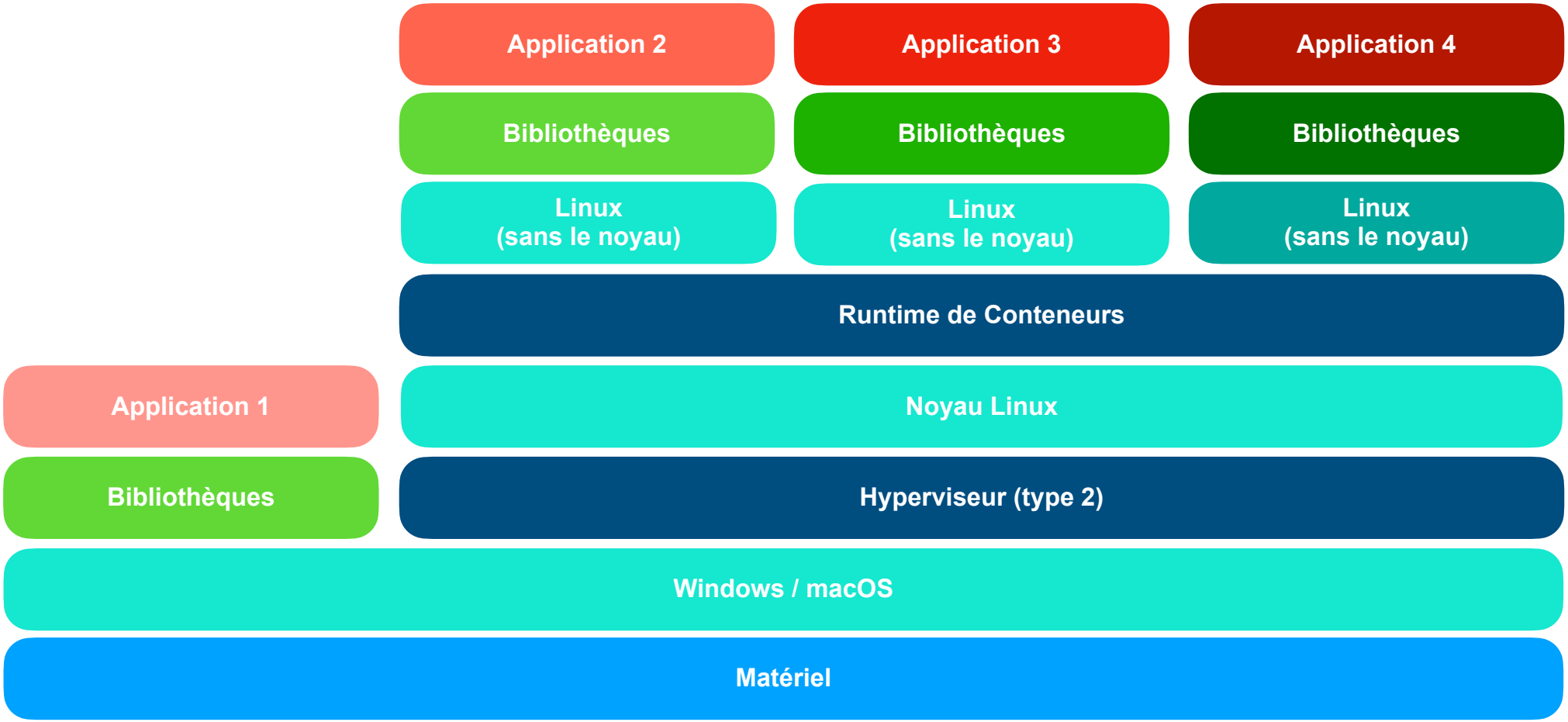


By V4711 - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=37649553

Conteneurisation (sur Linux)



Conteneurisation (sur Windows, macOS)



Conteneurisation : support système avec Linux

- Isolation du processus (namespace)
  - Limite ce que peut voir le processus
  - `clone()` est utilisé à la place de `fork()`

- Limitation des ressources
  - Limite ce que peut utiliser le processus
  - Système de fichiers virtuel (`cgroupfs`) monté généralement dans `/sys/fs/cgroup/`

- Isolation du système de fichiers
  - Changement du répertoire racine
  - `chroot()` ou `pivot_root()`