NTNU

Nettverksprogrammering

Programmeringsspråk og tråder

Ole C. Eidheim

January 9, 2025

Department of Computer Science

Oversikt

Øving P1

Sammenligning av C++, Rust og Java

Prosesser og tråder

Øving P1

- Finn alle primtall mellom to **gitte** tall ved hjelp av et **gitt** antall tråder.
  - Skriv til slutt ut en **sortert** liste av alle primtall som er funnet
  - Pass på at de ulike trådene får omtrent like mye arbeid
  - **Valgfritt programmeringsspråk**, men bruk gjerne et programmeringsspråk dere ikke har prøvd før (ikke Python), eller for de som vil ha litt extra utfordring: Rust eller C++
    - De som vil bruke Rust eller C++ kan ta utgangspunkt i threads

---

- Største primtall funnet (2016): $2^{74207281} - 1$
  - Great Internet Mersenne Prime Search
    - 348 708 000 GFLOP/sec (januar 2017)
    - Ole's Mac: 640 GFLOP/sec (grafikkortet)

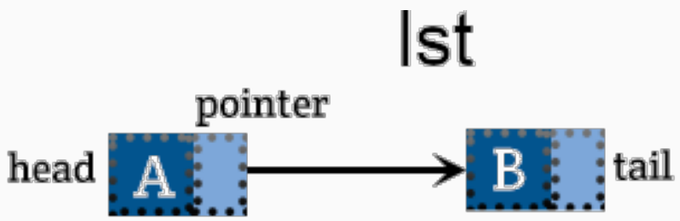
Oversikt

Øving P1

Sammenligning av C++, Rust og Java

Prosesser og tråder

Repitisjon: vector/array, list og iteratorer - iteratorer generaliserer lesing og skriving til konteinere




```cpp
#include <iostream>
#include <list>
#include <vector>

using namespace std;

int main() {
  {
    vector<char> vec = {'A', 'B'};
    auto it = vec.begin();
    cout << *it;          // Output: A
    it++;
    cout << *it << endl; // Output: B
  }
  {
    list<char> lst = {'A', 'B'};
    auto it = lst.begin();
    cout << *it;          // Output: A
    it++;
    cout << *it << endl; // Output: B
  }
}
```

Sammenligning av Java, C++ og Rust

- Eksempel: iterator-invalidation
  - Java gjør at du slipper å tenke på levetid, men det er lett å skrive andre typer feil
  - C++ har i utgangspunktet få begrensninger, og det er svært lett for uerfarne programmerere å gjøre feil
    - Derimot jobbes det mot bedre statiske sjekker som finner vanlige feil som nye programmerere gjør (se eksempel)
  - Rust setter strenge begrensninger på hvordan du kan skrive kode, og beskytter deg fra å gjøre vanlige feil
  - Men kan være vanskeligere å lese logikken i koden

Sammenligning av Java, C++ og Rust

- Eksempel: iterator-invalidation
  - Java gjør at du slipper å tenke på levetid, men det er lett å skrive andre typer feil
  - C++ har i utgangspunktet få begrensninger, og det er svært lett for uerfarne programmerere å gjøre feil
    - Derimot jobbes det mot bedre statiske sjekker som finner vanlige feil som nye programmerere gjør (se eksempel)
  - Rust setter strenge begrensninger på hvordan du kan skrive kode, og beskytter deg fra å gjøre vanlige feil
    - Men kan være vanskeligere å lese logikken i koden
- Både C++ og Rust er systemprogrammeringsspråk som genererer svært kjappe og minneeffektive program og programvarebiblioteker
  - Rust er svært nytt men har noen nye idèer som er interessante
  - Stort sett alle program og programvarebiblioteker er per i dag indirekte eller direkte skrevet i C/C++
    - C er et lavnivå programmeringsspråk, C++ er en utvidelse av C som gjør at en kan skrive høynivå kode på en enklere måte

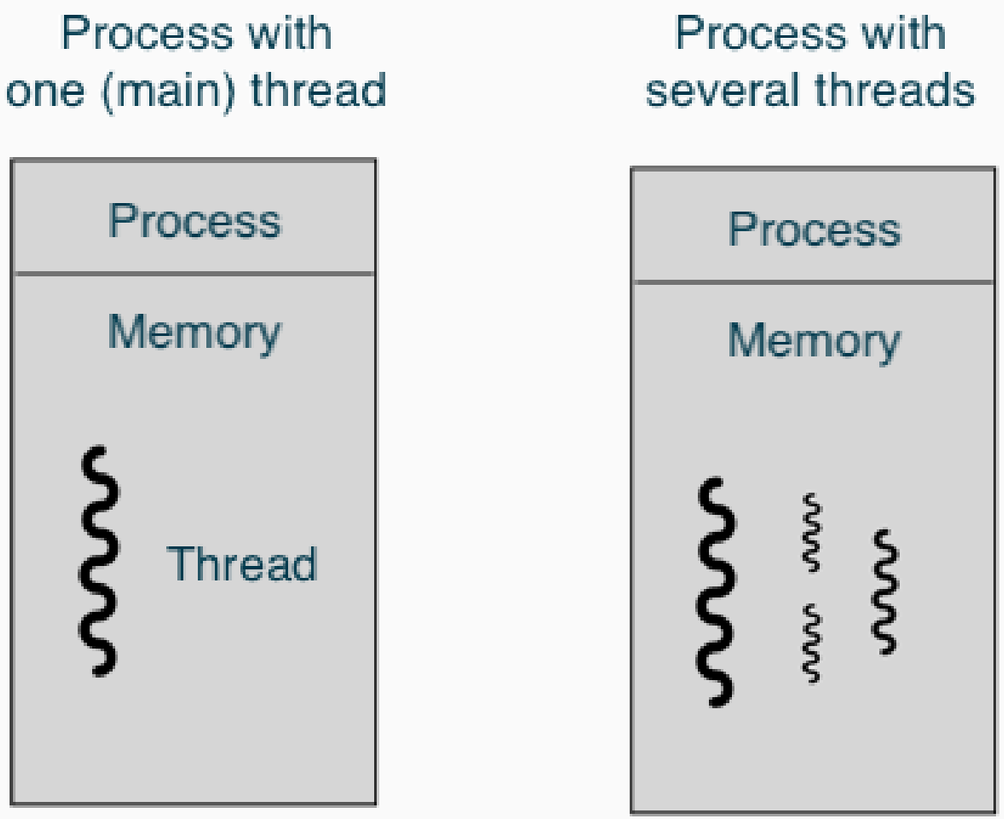
Oversikt

Øving P1

Sammenligning av C++, Rust og Java

Prosesser og tråder

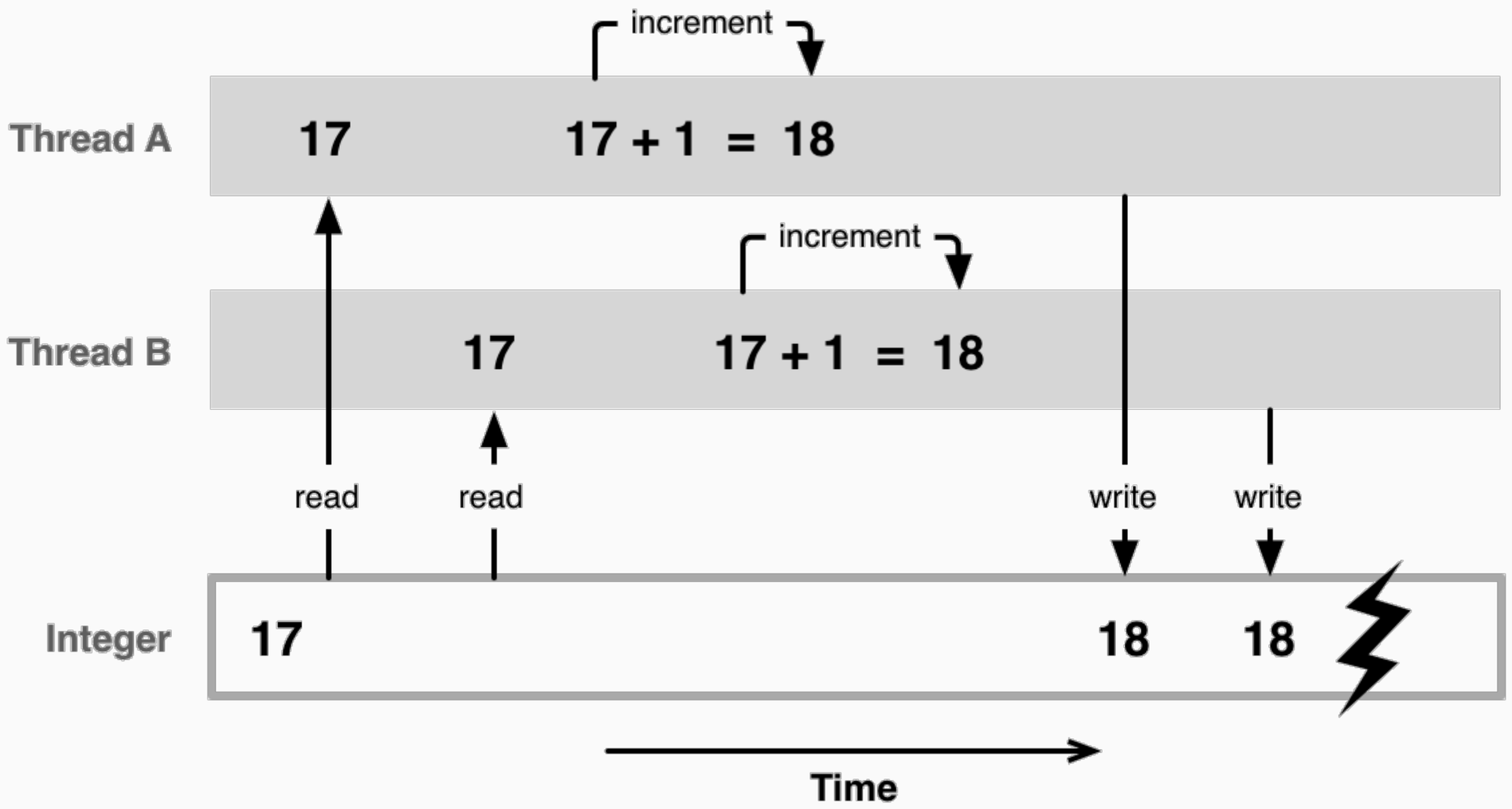
Prosesser og tråder



Trådeksempel i C++: enkel demonstrasjon av bruk av felles data

```cpp
#include <iostream>
#include <thread>

using namespace std;

int main() {
  int sum = 0;

  thread t1([&sum] {
    for (int i = 0; i < 1000; i++)
      sum++;
  });
  thread t2([&sum] {
    for (int i = 0; i < 1000; i++)
      sum++;
  });

  t1.join();
  t2.join();

  cout << sum << endl;
}
```

Samtidig lesing og skriving



Trådeksempel i C++, forbedret 1

```cpp
#include <iostream>
#include <mutex>
#include <thread>

using namespace std;

int main() {
  int sum = 0;
  mutex sum_mutex; // Used to make sure that only one thread accesses sum at any time

  thread t1([&sum, &sum_mutex] {
    for (int i = 0; i < 1000; i++) {
      sum_mutex.lock(); // If sum_mutex is already locked, wait until unlocked, then lock
      sum++;
      sum_mutex.unlock(); // sum_mutex can now be locked elsewhere
    }
  });
  thread t2([&sum, &sum_mutex] {
    for (int i = 0; i < 1000; i++) {
      sum_mutex.lock(); // If sum_mutex is already locked, wait until unlocked, then lock
      sum++;
      sum_mutex.unlock(); // sum_mutex can now be locked elsewhere
    }
  });

  t1.join();
  t2.join();

  cout << sum << endl;
}
// Output: 2000
```

Trådeksempel i C++, forbedret 2

```cpp
#include <iostream>
#include <mutex>
#include <thread>

using namespace std;

int main() {
  int sum = 0;
  mutex sum_mutex; // Used to make sure that only one thread accesses sum at any time

  thread t1([&sum, &sum_mutex] {
    for (int i = 0; i < 1000; i++) {
      unique_lock<mutex> lock(sum_mutex); // Locks sum_mutex
      sum++;
      // Unlocks sum_mutex when lock is destroyed at end of scope
    }
  });
  thread t2([&sum, &sum_mutex] {
    for (int i = 0; i < 1000; i++) {
      unique_lock<mutex> lock(sum_mutex); // Locks sum_mutex
      sum++;
      // Unlocks sum_mutex when lock is destroyed at end of scope
    }
  });

  t1.join();
  t2.join();

  cout << sum << endl;
}
// Output: 2000
```

Trådeksempel i C++, forbedret 3

Fra cpp-thread-safety-analysis:

```
class Main { // Thread Safety Analysis only applies to classes
public:
  int sum GUARDED_BY(sum_mutex) = 0; // Extra annontation to restrict access to sum
  Mutex sum_mutex;

  Main() {
    thread t1([this] { // Captures current instance (this) for access to sum and sum_mutex
      for (int i = 0; i < 1000; i++) {
        LockGuard lock(sum_mutex); // Must lock the mutex to access sum
        sum++;
      }
    });
    thread t2([this] { // Captures current instance (this) for access to sum and sum_mutex
      for (int i = 0; i < 1000; i++) {
        LockGuard lock(sum_mutex); // Must lock the mutex to access sum
        sum++;
      }
    });

    t1.join();
    t2.join();

    LockGuard lock(sum_mutex); // Must lock the mutex to access sum
    cout << sum << endl;
  }
};

int main() {
  Main();
} // Output: 2000
```

Trådeksempel i Rust

```rust
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
  // Arc: thread-safe reference counted object.
  // Mutex: data and mutex combined, where the data cannot be access without locking the mutex
  let sum_mutex_arc = Arc::new(Mutex::new(0));

  let sum_mutex_arc_copy = sum_mutex_arc.clone();
  let t1 = thread::spawn(move || {
    for i in 0..1000 {
      // Access the data by locking the Mutex object
      let mut sum_locked = sum_mutex_arc_copy.lock().unwrap();
      *sum_locked += 1; // Access the locked data throught the operator*
      // The Mutex object is unlocked at end of scope
    }
  });
  let sum_mutex_arc_copy = sum_mutex_arc.clone();
  let t2 = thread::spawn(move || {
    for i in 0..1000 {
      // Access the data by locking the Mutex object
      let mut sum_locked = sum_mutex_arc_copy.lock().unwrap();
      *sum_locked += 1; // Access the locked data throught the operator*
      // The Mutex object is unlocked at end of scope
    }
  });

  t1.join();
  t2.join();
  // Cannot access data without calling lock(), even though locking is unnecessary.
  println!("{}", *sum_mutex_arc.lock().unwrap())
} // Output: 2000
```

Deadlocks: uendelig venting på at en mutex skal bli låst opp

Rust beskytter deg ikke mot deadlocks, men det gjør C++ Thread Safety Analysis

```
class Main { // Thread Safety Analysis only applies to classes
public:
  int sum GUARDED_BY(sum_mutex) = 0; // Extra annontation to restrict access to sum
  Mutex sum_mutex;

  Main() {
    thread t1([this] {
      for (int i = 0; i < 1000; i++) {
        LockGuard lock1(sum_mutex);
        LockGuard lock2(sum_mutex); // Warning: sum_mutex is already locked
        sum++;
      }
    });

    t1.join();

    LockGuard lock(sum_mutex);
    cout << sum << endl;
  }
};

int main() {
  Main();
} // Output: 2000
```

Deadlocks: uendelig venting på at en mutex skal bli låst opp

Rust beskytter deg ikke mot deadlocks, men det gjør C++ Thread Safety Analysis

```
class Main { // Thread Safety Analysis only applies to classes
public:
  int sum GUARDED_BY(sum_mutex) = 0; // Extra annontation to restrict access to sum
  Mutex sum_mutex;

  Main() {
    thread t1([this] {
      for (int i = 0; i < 1000; i++) {
        sum_mutex.lock(); // Locks mutex instead of using lock guard
        sum++;
        // Warning: sum_mutex is still locked
      }
    });

    t1.join();

    LockGuard lock(sum_mutex);
    cout << sum << endl;
  }
};

int main() {
  Main();
} // Output: 2000
```

Tråder: overhead

Ikke alltid fornuftig å dele oppgaver i tråder: