Synchronization

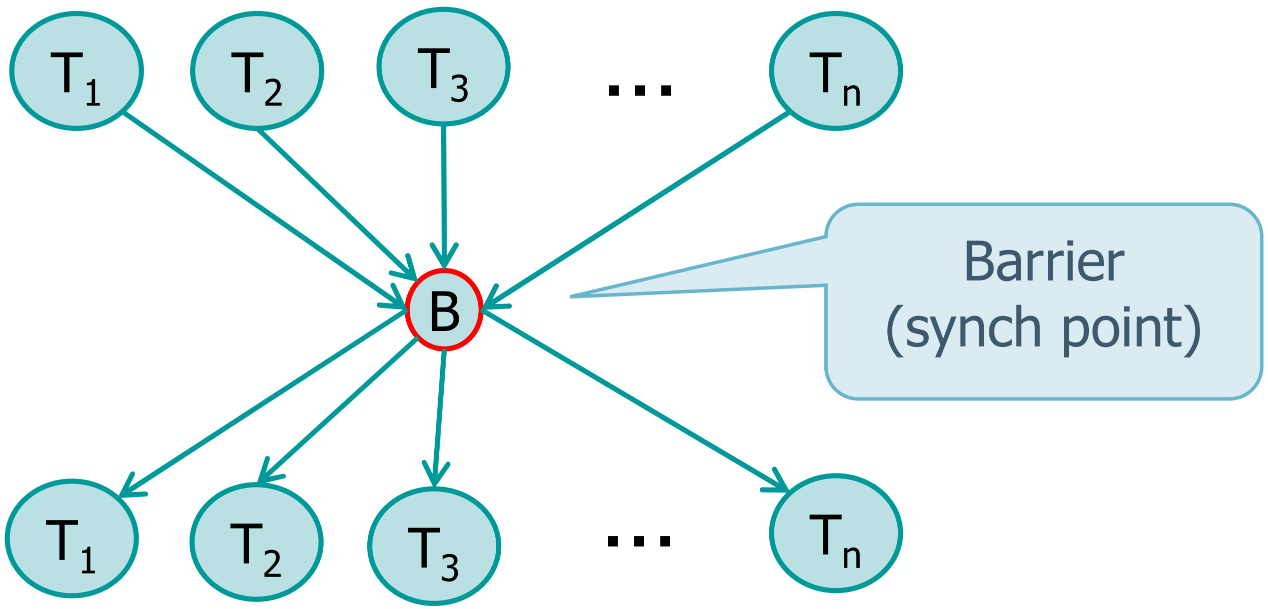
Barriers

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

License Information

Barriers

- Barriers can be used to coordinate multiple threads working in parallel
  - A barrier allows each thread to wait until all cooperating threads have reached the same point, and then continue executing from there



Barriers

- Barriers generalize the t.join() function
  - The function **t.join()** acts as a barrier to allow **one** thread to wait until another thread completes its processing
  - Barriers allow **an arbitrary number** of threads to wait until all of the threads have completed their processing
  - The threads don't have to exit, as they can continue working after all threads have reached the barrier

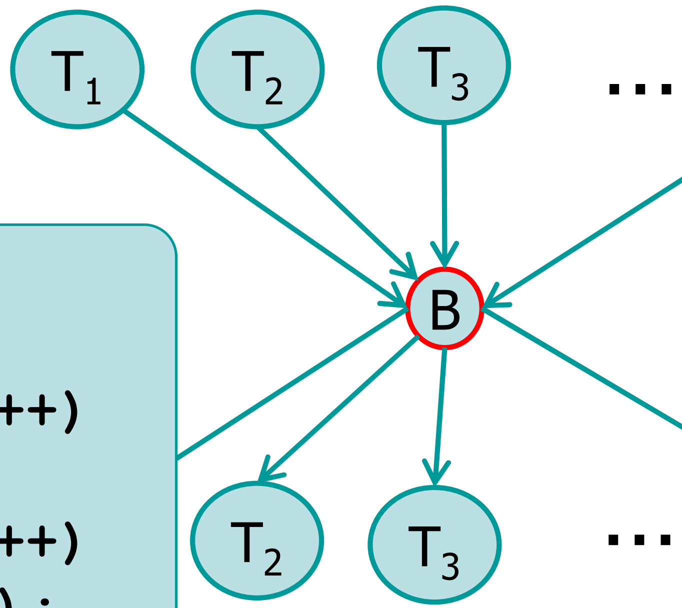
Trivial solution

It uses too many semaphores

- A possible trivial solution
  - Use one semaphore for each thread T_i
  - It implement one for the extra process B (for the barrier) using one more semaphore

The barrier process waits n times on its semaphores and then wakes-up all threads T_i

```
B (pseudo-code)

for (i=0; i<n; i++)
  wait (semB);
for (i=0; i<n; i++)
  signal (sem[i]);
```



T_i (pseudo-code)

```
signal (semB);
wait (sem[i]);

...
```

Each T_i wakes-up the barrier process and wait on its own semaphore

Barriers in C++

- In C++, synchronization barriers can be implemented using several primitives, with notable additions in C++20
  - A std::latch is a one-time synchronization primitive
    - It counts down from an initial value set in its constructor
    - Ideal when threads need to wait for a specific number of tasks to complete before proceeding

Methods	Meaning
wait()	To wait at the barrier until the counter reaches zero.
count_down()	Decrement the counter without waiting.

Barriers in C++

- A std::barrier is similar to std::latch, but **reusable**
  - After the counter reaches zero, it resets to its initial value, allowing for repeated synchronization
  - Suitable for phased computations where threads need to synchronize multiple times

Methods	Meaning
arrive_and_wait()	To wait at the barrier.
arrive_and_drop()	To remove a thread from the barrier without waiting.

Barriers in C++

- A std::flex_barrier is an extension of std::barrier, allowing **the execution of a function** when the counter reaches zero
  - This function can be specified during barrier construction.

Example

Main program calling the threads

```
#include <iostream>
#include <barrier>
#include <thread>
#include <vector>

std::mutex mtx_out;
void worker(std::barrier<> &);

int main() {
  const int numThreads = 10;
  std::barrier<> barrier(numThreads);
  std::vector<std::thread> threads;
  for (int i = 0; i < numThreads; ++i) {
    threads.emplace_back(worker, std::ref(barrier));
  }
  for (auto &thread : threads) {
    thread.join();
  }
    return 0;
}
```

Used for a general I/O critical section

Define the barrier for 10 trhreads

Example

Use the barrier to synch numThreads **once**

```
void worker(std::barrier<> &barrier) {
  // Perform some work
  mtx_out.lock();
  std::cout << "Thread performing work...\n";
  mtx_out.unlock();

  // Wait for all threads to reach this point
  barrier.arrive_and_wait();

  // Continue after all threads have synchronized
  mtx_out.lock();
  std::cout << "Thread continuing after barrier.\n";
  mtx_out.unlock();

  return;
}
```

The mutex protec tcout

The mutex protect cout

Example

Use the barrier to synch iterations **times**

```
void worker(std::barrier<> &barrier, int id,
  int iterations) {
  for (int i = 0; i < iterations; ++i) {
    mtx_out.lock();
    cout << "Thread " << id << " it " << i << endl;
    mtx_out.unlock();

    barrier.arrive_and_wait();

    mtx_out.lock();
    cout << "Thread " << id <<
            " after barrier on " << i << endl;
    mtx_out.unlock();
  }
  return;
}
```

The barrier does not have to be re-initialized

The mutex protect cout

Conclusions

- Barriers are used to coordinate multiple threads working in parallel
  - You want all threads to wait until everyone has arrived at a certain point
  - A simple semaphore would do the exact opposite, i.e., each thread would keep running and the last one will go to sleep

Exercise 01

Acyclic barrier

- Suppose barrier constructs do not exist
  - Re-implement them using only **one** semaphore **and** one mutex

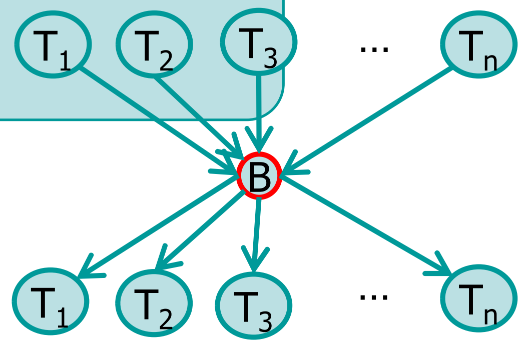
Main

```
std::barrier<> barrier(numThreads);
std::vector<std::thread> threads;
for (int i = 0; i < numThreads; ++i) {
  threads.emplace_back(worker, std::ref(barrier));
}
for (auto &thread : threads) {
  thread.join();
}
```

Threads (acyclic behavior)

```
void worker(std::barrier<> &barrier) {
  ...
  barrier.arrive_and_wait();
  ...
}
```

Synchronization point among all threads



Solution 01

```
#include <mutex>
#include <semaphore>
#include <thread>
#include <vector>
#include <iostream>

const int numThreads = 10;
std::mutex mtx;
std::mutex mtx_out;
std::counting_semaphore<numThreads> sem{0};
int count;    // Total number of threads
int arrived; // Number of threads that have arrived

void worker(int);
```

Solution 01

```
int main() {
  count = numThreads;
  arrived = 0;

  std::vector<std::thread> threads;
  for (int i = 0; i < numThreads; ++i) {
    threads.emplace_back(worker, i);
  }

  for (auto &thread : threads) {
    thread.join();
  }

  return 0;
}
```

Solution 01

```
void worker(int id) {
  mtx_out.lock();
  std::cout << "Thread " << id << " work..." << std::endl;
  mtx_out.unlock();
  mtx.lock();
  arrived++;
  if (arrived == count) {
    for (int i=0; i<count; ++i) sem.release();
  }
  mtx.unlock();
  sem.acquire();
  mtx_out.lock();
  std::cout << "Thread " << id << " after barrier." << std::endl;
  mtx_out.unlock();
  return;
}
```

Protect counter

Un-protect counter

Solution 01

Solution with **turnstile**

```
void worker(int id) {
  mtx_out.lock();
  std::cout << "Thread " << id << " work..." << std::endl;
  mtx_out.unlock();
  mtx.lock();
  arrived++;
  if (arrived == count)
    sem.release();
  mtx.unlock();
  sem.acquire();
  sem.release();
  mtx_out.lock();
  std::cout << "Thread " << id << " after barrier." << std::endl;
  mtx_out.unlock();
  return;
}
```

Protect counter

Un-protect counter

Turnstile

One **extra** sem_post is done (pay attention to cycling threads)

Exercise 02

Cyclic barrier

- Suppose barrier constructs do not exist
  - Re-implement them using only **one** semaphore **and** one mutex

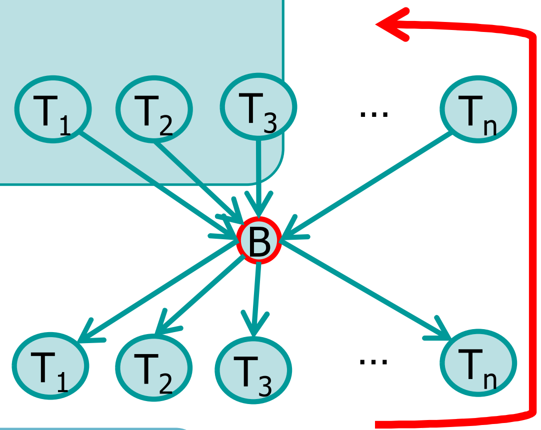
Main

```
std::barrier<> barrier(numThreads);
std::vector<std::thread> threads;
for (int i = 0; i < numThreads; ++i) {
  threads.emplace_back(worker, std::ref(barrier));
}
for (auto &thread : threads) {
  thread.join();
}
```

Threads (cyclic behavior)

```
void worker(std::barrier<> &barrier) {
  for (int i = 0; i < iterations; ++i) {
    ...
    barrier.arrive_and_wait();
    ...
  }
}
```

Synchronization point among all threads



Buggy Solution

Buggy solution

```
void worker(int id) {
  for (int i=0; i<iterations; i++) {
    mtx_out.lock();
    std::cout << "Thread " << id << " work..." << std::endl;
    mtx_out.unlock();
    mtx.lock();
    arrived++;
    if (arrived == count) {
      arrived = 0;
      for (int i=0; i<count; ++i) sem.release();
    }
    mtx.unlock();
    sem.acquire();
    mtx_out.lock();
    std::cout << "Thread " << id << " continuing after barrier." << std::endl;
    mtx_out.unlock();
  }
  return;
}
```

Last threads awakes all

Waiting point for all threads

A fast threads can cycle more than once !

Solution

Correct solution
(part I)

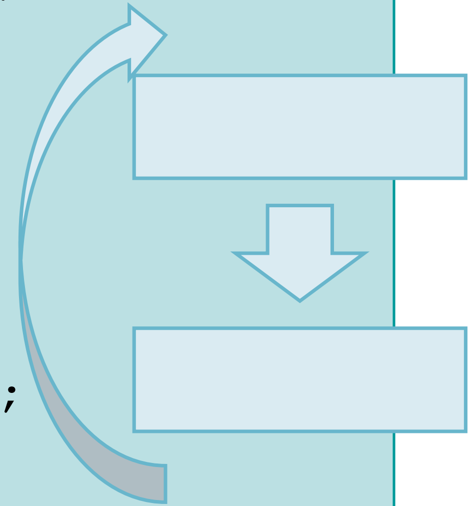
```
void worker(int id) {
  for (int i=0; i<iterations; i++) {
    mtx_out.lock();
    std::cout << "Thread " << id <<
        " performing work..." << std::endl;
    mtx_out.unlock();
    mtx.lock();
    arrived++;
    if (arrived == count) {
      for (int i=0; i<count; ++i) sem1.release();
    }
    mtx.unlock();
    sem1.acquire();
    mtx_out.lock();
    std::cout << "Thread " << id <<
      " moving from B1 to B2." << std::endl;
    mtx_out.unlock();
```

Barrier #1

Correct solution (part II)

Barrier #2

```
    mtx.lock();
    arrived--;
    if (arrived == 0) {
      for (int i=0; i<count; ++i) sem2.release();
    }
    mtx.unlock();
    sem2.acquire();
    mtx_out.lock();
    std::cout << "Thread " << id <<
      " continuing after barrier." << std::endl;
    mtx_out.unlock();
  }

  return;
}
```



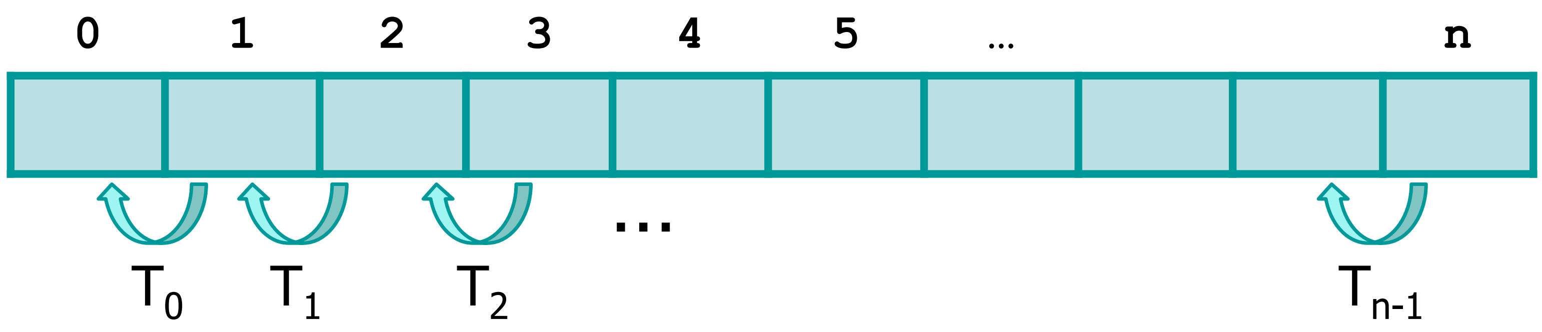
Exercise 03

Concurrent Bubble-sort

- Write a version of the exchange (bubblesort) sorting algorithm) as follows
  - A static array include n integer values
  - We want to ort it using n identical threads
  - Each thread is in charge of sorting two adjacent elements
    - Thread 0 sort elements 0 and 1
    - Thread 1 sort elements 1 and 2
    - …
    - Thread n-1 sort elements n-1 and n

Exercise 03

- Each thread
  - Compare the two elements it deals with, and exchange them if they are not in the correct order
  - Once their work is finished, all the threads wait for each-other, and if
    - All the elements are correctly ordered, the program terminates
    - Otherwise, all threads are run again to make a new series of exchanges

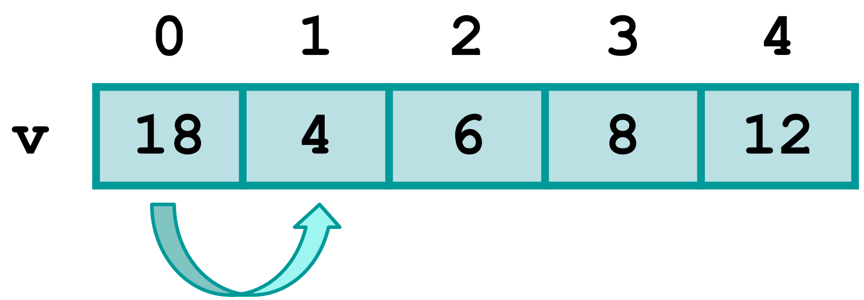


Exercise 03

- Each thread
  - Compare the two elements it deals with, and exchange them if they are not in the correct order
  - Once their work is finished, all the threads wait for each-other, and if
    - All the elements are correctly ordered, the program terminates
    - Otherwise, all threads are run again to make a new series of exchanges

As the order in which all swaps are performed is not defined (inner iteration) the number of necessary outer iterations is upper bounded by n

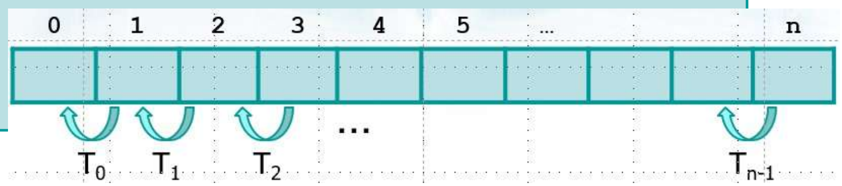
```
for (i=0; i<n-1; i++)
  for (j=0; j<n-i-1; j++)
    if (v[j] > v[j+1])
      swap (v, i, j+1);
```



Solution

Main (Estract)
Part I

```
#include <thread>
#include <vector>
#include <iostream>
#include <barrier>
#include <mutex>

void sorter (int *vet, int vet_size, int i,
  std::vector<std::mutex> &me, std::barrier<> &barrier,
  std::mutex &mtx_out, bool &sorted, bool &all_ok);

int main(int argc, char** argv) {
  if (argc != 2) {
    std::cout << "Syntax: " << argv[0] << " vet_size\n";
    return 1;
  }
```



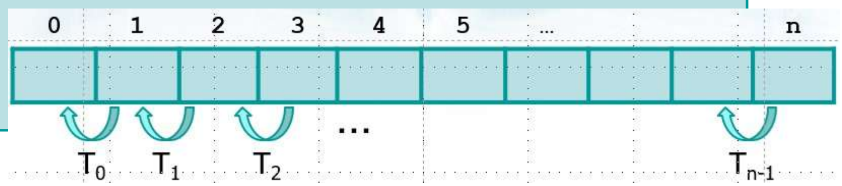
Solution

Main (Estract) Part II

```
const int vet_size = std::stoi(argv[1]);
const int num_threads = vet_size - 1;
std::mutex mtx_out;
std::barrier<> barrier(num_threads);
std::vector<std::mutex> me(vet_size);
int *vet = new int[vet_size];
bool sorted = false;
bool all_ok = true;

std::srand(std::time(nullptr));

for (int i=0; i<vet_size; i++) {
  vet[i] = std::rand() % 1000;
  std::cout << vet[i] << " ";
}
std::cout << "\n";
```

Create a mutex for each element of the vector

Fill the vector with random numbers



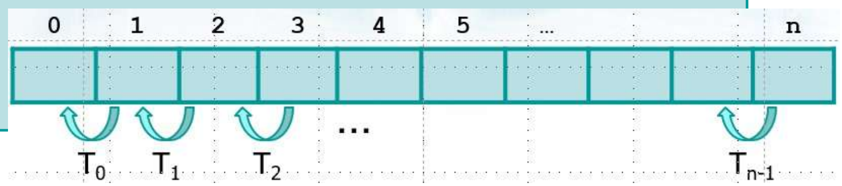
Solution

Main (Estract)
Part III

```
std::vector<std::thread> threads;
for (int i=0; i<num_threads; i++) {
  threads.emplace_back(sorter, vet, vet_size, i,
    std::ref(me), std::ref(barrier),
    std:ref(mtx_out), std::ref(sorted), std::ref(all_ok));
}

for (auto& thread : threads) {
  thread.join();
}

delete[] vet;

return 0;
}
```

Run threads

Wait for threads



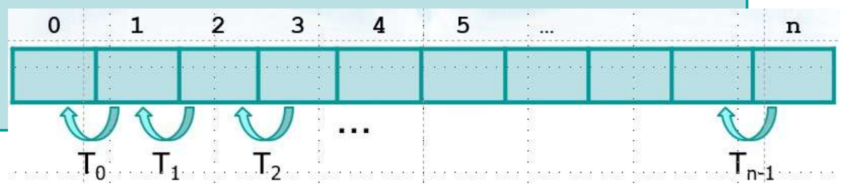
Solution

Worker
Part 1

```
void sorter (int *vet, int vet_size, int i,
  std::vector<std::mutex> &me, std::barrier<> &barrier,
  std::mutex &mtx_out, bool &sorted, bool &all_ok) {
  while (!sorted) {
    {
      std::lock_guard<std::mutex> lock1(me[i]);
      std::lock_guard<std::mutex> lock2(me[i+1]);
      if (vet[i] > vet[i+1]) {
        mtx_out.lock();
        mtx_out.unlock();
        int tmp = vet[i];
        vet[i] = vet[i + 1];
        vet[i + 1] = tmp;
        all_ok = false;
      }
    }
```

Acquire mutexes for two contiguous elements

Swap elements if necessary



Solution

Worker
Part II

```
    barrier.arrive_and_wait();
    if (i==0) {
      for (int i=0; i<vet_size; i++) {
        std::cout << vet[i] << " ";
      }
      std::cout << std::endl;
      if (all_ok)
        sorted = true;
      else
        all_ok = true;
    }
    barrier.arrive_and_wait();
  }
  return;
}
```

If no exchanges have been done, stop the process