Parallel Computing

IT00CD91-3005, Spring 2025

Åbo Akademi

Lecture 3: Shared Memory Programming – **pthreads, std::threads, OpenMP**

by Alireza Olama

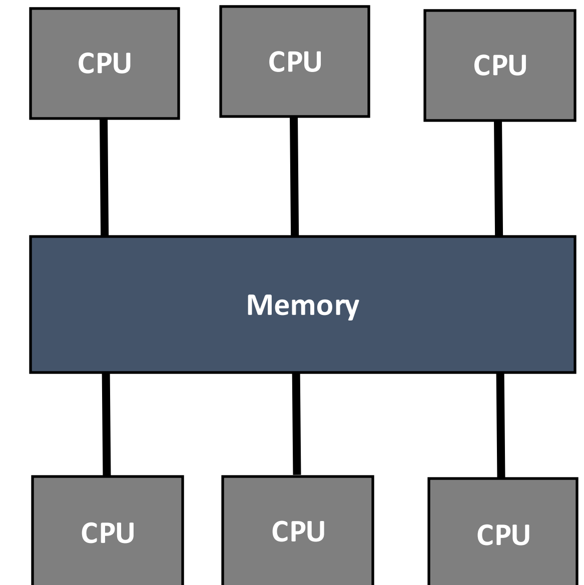
Shared Memory Systems

The entire system's **memory is shared** among all CPU **cores**.

Shared memory helps to **coordinate** the work of cores.

implicit communication

collaboration



Programming Strategies

- **Multi-threading**
- **Multi-processing**

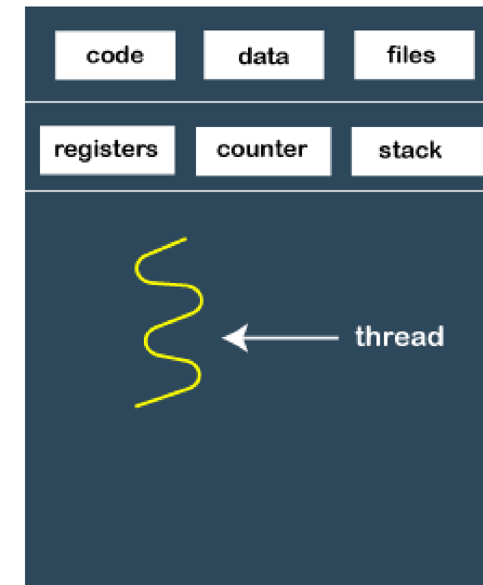
Processes and Threads

A **process** is an instance of a running (or suspended) program.

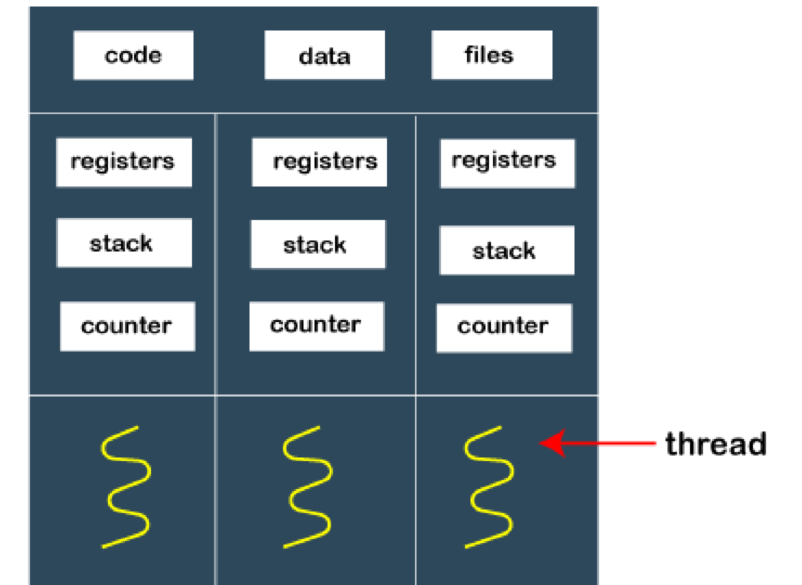
Processes and Threads

A **process** is an instance of a running (or suspended) program.

A Thread (of control) is independent sequences of instructions within a program that can execute concurrently, sharing the same resources of the process. Also called light-weight processes.



Single-threaded process



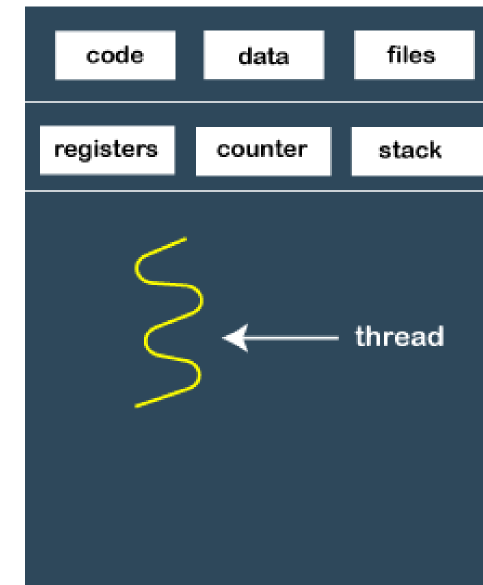
Multi-threaded process

Processes and Threads

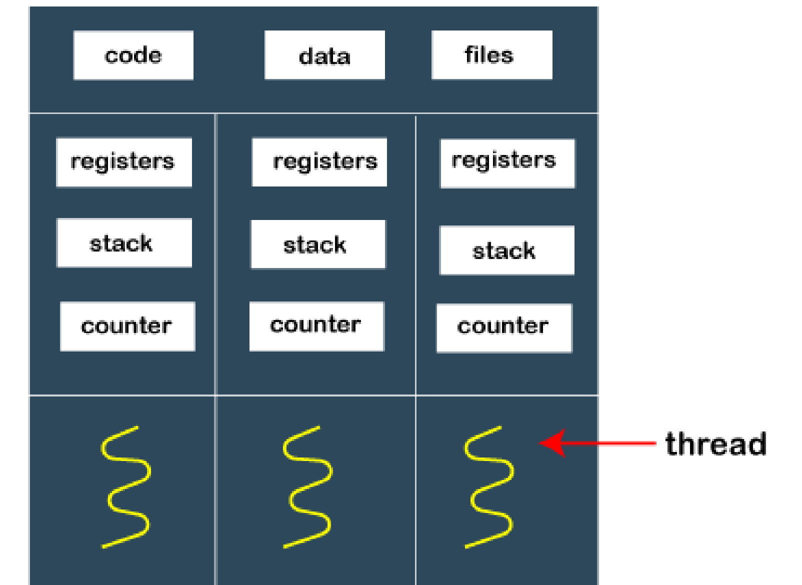
A **process** is an instance of a running (or suspended) program.

A Thread (of control) is independent sequences of instructions within a program that can execute concurrently, sharing the same resources of the process. Also called light-weight processes.

A single process may have multiple threads of control



Single-threaded process



Multi-threaded process

Processes and Threads

A **process** is an instance of a running (or suspended) program.

A Thread (of control) is independent sequences of instructions within a program that can execute concurrently, sharing the same resources of the process. Also called light-weight processes.

Single-threaded process

Multi-threaded process

A single process may have multiple threads of control

How to program shared memory systems?

1. pthreads
2. std::threads – **C++11**
3. **OpenMP – The focus of this class**

OpenMP®

Parallel Programming With `POSIX Threads`

POSIX Threads – `pthreads`

POSIX: Portable **O**perating **S**ystem **I**nterface – interface to Operating system utilities.

A set of APIs and specifications to ensure compatibility between different UNIX-like operating systems.

POSIX Threads – `pthreads`

POSIX: **P**ortable **O**perating **S**ystem **I**nterface – interface to Operating system utilities.

A set of APIs and specifications to ensure compatibility between different UNIX-like operating systems.

`PThreads`: The POSIX threading interface

- System calls to create and synchronize threads
- Should be relatively uniform across **UNIX-like OS platforms**

POSIX Threads – `pthreads`

POSIX: Portable **O**perating **S**ystem **I**nterface – interface to Operating system utilities.

A set of APIs and specifications to ensure compatibility between different UNIX-like operating systems.

`PThreads`: The POSIX threading interface

- System calls to create and synchronize threads
- Should be relatively uniform across **UNIX-like OS platforms**

Pthreads contain support for

- Creating parallelism
- Synchronization
- No explicit support for communication, because shared memory is implicit; a pointer to shared data is passed to a thread.

Creating POSIX Threads

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

```
pthread_create(&thread, NULL, my_thread_function, (void*)msg);
```

- **thread**: a pointer to a **pthread_t** variable where the thread id will be stored.
- **attr**: pointer to thread attributes (or **NULL** for default)
  - Standard default values obtained by passing a **NULL** pointer
  - Sample attributes: minimum stack size, priority
- **start_routine**: pointer to the function the thread will run
- **arg**: argument passed to the thread function (can be **NULL**)
- returns 0 on success
- non-zero error code on failure

POSIX Threads Hello World!

Compilation with gcc

```
gcc -pthread -o hello_threads hello_threads.c
```

Compilation with CMAKE

```
cmake_minimum_required(VERSION 3.30)
project(HelloThreads C)

set(CMAKE_C_STANDARD 11)

add_executable(hello_threads hello_threads.c)
target_link_libraries(hello_threads pthread)
```

```c
void* SayHello(void *foo) {
    printf("Hello, world!\n");
    return NULL;
}

int main() {
    pthread_t threads[16];
    int tn;

    for (tn = 0; tn < 16; tn++) {
        pthread_create(&threads[tn], NULL, SayHello, NULL);
    }

    for (tn = 0; tn < 16; tn++) {
        pthread_join(threads[tn], NULL);
    }

    return 0;
}
```

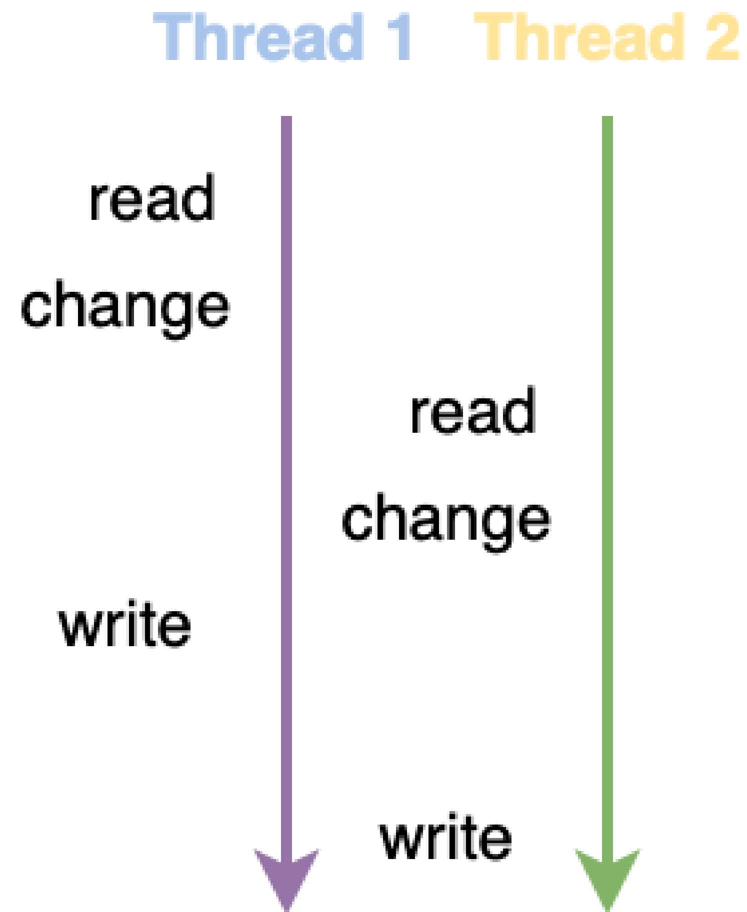
Recall Data Race

- Problem is a race condition on variable **`counter`** in the program
- A **race condition** or **data race** occurs when:
  - Two processors or (two threads) access **the same variable**, and at least one does a **write**.
  - The accesses are concurrent (**not synchronized**) so they could happen simultaneously.

```c
int counter = 0;  // Shared global variable

void* increment(void* arg) {
    for (int i = 0; i < 100000; i++) {
        counter++;  // Not thread-safe: potential data race
    }
    return NULL;
}

int main() {
    pthread_t t1, t2;

    pthread_create(&t1, NULL, increment, NULL); // Thread 1 increments counter
    pthread_create(&t2, NULL, increment, NULL); // Thread 2 also increments counter

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("Final counter value: %d\n", counter);
    return 0;
}
```

CRITICAL SECTION

Recall Data Race




```c
int counter = 0; // Shared global variable

void* increment(void* arg) {
    for (int i = 0; i < 100000; i++) {
        counter++; // Not thread-safe: potential data race
    }
    return NULL;
}

int main() {
    pthread_t t1, t2;

    pthread_create(&t1, NULL, increment, NULL); // Thread 1 increments counter
    pthread_create(&t2, NULL, increment, NULL); // Thread 2 also increments counter

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("Final counter value: %d\n", counter);
    return 0;
}
```

CRITICAL SECTION

```
mov eax, [counter]
add eax, 1
mov [counter], eax
```

Recall Data Race

what is going on exactly?

- `counter = 42`
- **thread 1** loads 42
- **thread 2** also loads 42
- **thread 1 increments to 43 and stores 43**
- **thread 2 increments its own copy of 42 to 43 and stores it**

- **counter is 43 instead of 44 → 1 increment loss**
- this can happen thousands of times
- **the final value may not be 200000**

```c
int counter = 0;  // Shared global variable

void* increment(void* arg) {
    for (int i = 0; i < 100000; i++) {
        counter++;  // Not thread-safe: potential data race
    }
    return NULL;
}

int main() {
    pthread_t t1, t2;

    pthread_create(&t1, NULL, increment, NULL); // Thread 1 increments counter
    pthread_create(&t2, NULL, increment, NULL); // Thread 2 also increments counter

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("Final counter value: %d\n", counter);
    return 0;
}
```

CRITICAL SECTION

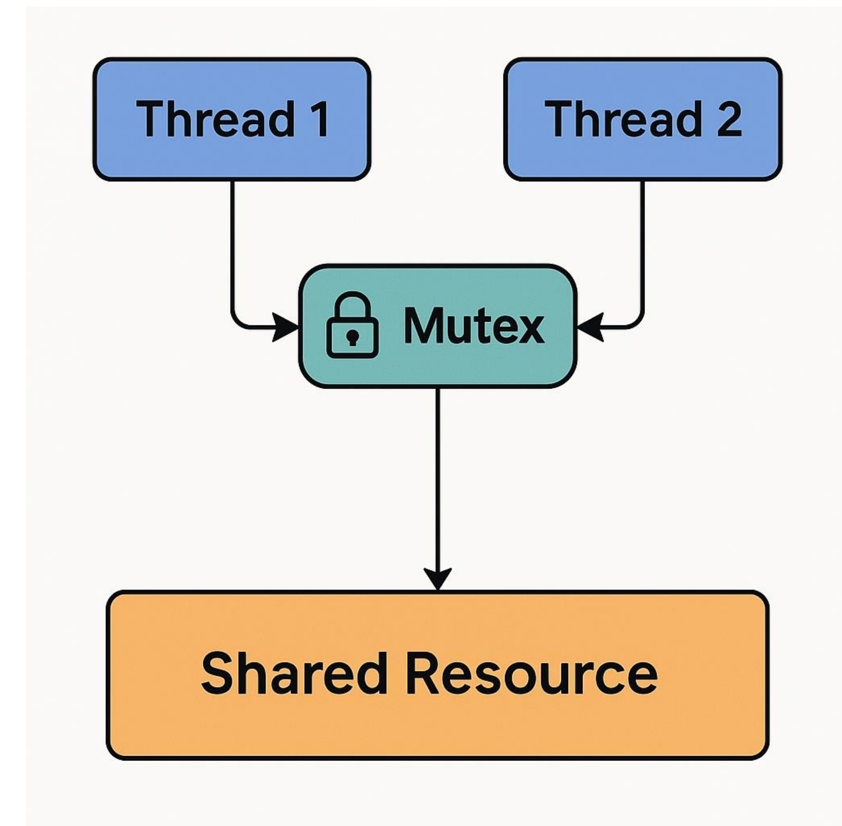
```
mov eax, [counter]
add eax, 1
mov [counter], eax
```

Basic Synchronization: Mutexes

- **Mutexes – mutual exclusion aka locks**
  - Threads are working mostly independently
  - Need to **access common data structures** (critical section)

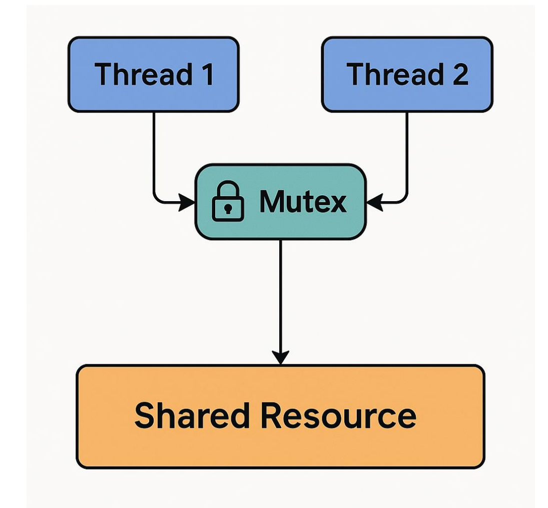
```
lock *l = alloc_and_init();  // Shared lock

acquire(l);                  // Acquire the lock
access_data();               // Safely access shared data
release(l);                  // Release the lock
```

  - Locks only affect processors using them:
    - If a thread accesses the data without doing the the acquire/release, locks by others will not help



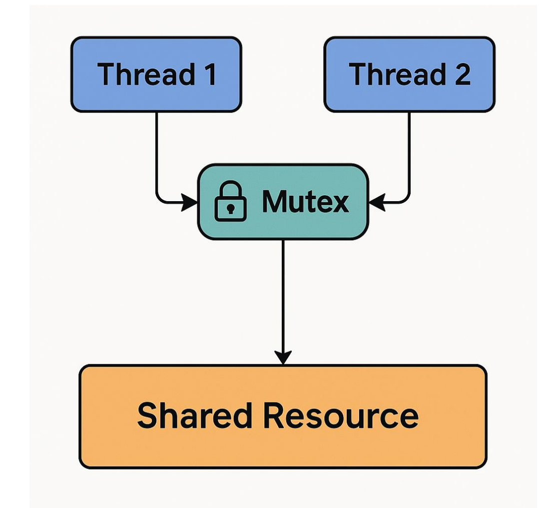
Basic Synchronization: Mutexes

- `Java, C++` and other languages have **lexically scoped synchronization**
  - The boundaries of the synchronization (lock acquire & release) are tied to the code block's structure in the source code.
  - When you enter (leave) a block, a lock is automatically acquired (released)
  - In `pthreads`, synchronization is managed manually.



Basic Synchronization: Mutexes

- `Java, C++` and other languages have **lexically scoped synchronization**
  - The boundaries of the synchronization (lock acquire & release) are tied to the code block's structure in the source code.
  - When you enter (leave) a block, a lock is automatically acquired (released)
  - In `pthreads`, synchronization is managed manually.




```java
class Counter {
    int value = 0;

    void increment() {
        synchronized(this) {
            value++;
        }
    }
}
```

```java
synchronized void increment() {
    value++;
}
```

```cpp
std::mutex m;
void f() {
    std::lock_guard<std::mutex> lock(m);
    // lock is acquired here

    // do work...

} // when this block ends, lock is released automatically
```

Java

C++

Basic Synchronization: Mutexes

Mutex initialization

```c
int counter = 0;  // Shared global variable
pthread_mutex_t lock;  // Mutex for protecting counter

void* increment(void* arg) {
    for (int i = 0; i < 100000; i++) {
        pthread_mutex_lock(&lock);     // Lock before accessing shared data
        counter++;
        pthread_mutex_unlock(&lock);   // Unlock after done
    }
    return NULL;
}

int main() {
    pthread_t t1, t2;

    pthread_mutex_init(&lock, NULL);  // Initialize the mutex

    pthread_create(&t1, NULL, increment, NULL);
    pthread_create(&t2, NULL, increment, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    pthread_mutex_destroy(&lock);  // Clean up

    printf("Final counter value: %d\n", counter);
    return 0;
}
```

Thread Safe

Basic Synchronization: Mutexes

Mutex initialization

Protecting Critical Section

```
int counter = 0;  // Shared global variable
pthread_mutex_t lock;  // Mutex for protecting counter

void* increment(void* arg) {
    for (int i = 0; i < 100000; i++) {
        pthread_mutex_lock(&lock);    // Lock before accessing shared data
        counter++;
        pthread_mutex_unlock(&lock);  // Unlock after done
    }
    return NULL;
}

int main() {
    pthread_t t1, t2;

    pthread_mutex_init(&lock, NULL);  // Initialize the mutex

    pthread_create(&t1, NULL, increment, NULL);
    pthread_create(&t2, NULL, increment, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    pthread_mutex_destroy(&lock);  // Clean up

    printf("Final counter value: %d\n", counter);
    return 0;
}
```

Thread Safe

Basic Synchronization: Mutexes

Mutex initialization

Protecting Critical Section

Deallocating a mutex

Thread Safe

```c
int counter = 0;  // Shared global variable
pthread_mutex_t lock;  // Mutex for protecting counter

void* increment(void* arg) {
    for (int i = 0; i < 100000; i++) {
        pthread_mutex_lock(&lock);    // Lock before accessing shared data
        counter++;
        pthread_mutex_unlock(&lock);  // Unlock after done
    }
    return NULL;
}

int main() {
    pthread_t t1, t2;

    pthread_mutex_init(&lock, NULL);  // Initialize the mutex

    pthread_create(&t1, NULL, increment, NULL);
    pthread_create(&t2, NULL, increment, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    pthread_mutex_destroy(&lock);  // Clean up

    printf("Final counter value: %d\n", counter);
    return 0;
}
```

Basic Synchronization: Mutexes

Mutex initialization

Protecting Critical Section

Multiple mutexes may be held, but can lead to problems

Thread 1	Thread 2
Lock(a)	Lock(b)
Lock(b)	Lock(a)

Deallocating a mutex

Thread Safe

```c
int counter = 0;  // Shared global variable
pthread_mutex_t lock;  // Mutex for protecting counter

void* increment(void* arg) {
    for (int i = 0; i < 100000; i++) {
        pthread_mutex_lock(&lock);     // Lock before accessing shared data
        counter++;
        pthread_mutex_unlock(&lock);   // Unlock after done
    }
    return NULL;
}

int main() {
    pthread_t t1, t2;

    pthread_mutex_init(&lock, NULL);  // Initialize the mutex

    pthread_create(&t1, NULL, increment, NULL);
    pthread_create(&t2, NULL, increment, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    pthread_mutex_destroy(&lock)  // Clean up

    printf("Final counter value: %d\n", counter);
    return 0;
}
```

Basic Synchronization: Mutexes

Mutex initialization

Protecting Critical Section

Multiple mutexes may be held, but can lead to problems

Thread 1		Thread 2
Lock(a)	deadlock	Lock(b)
Lock(b)		Lock(a)

Deallocating a mutex

Thread Safe

```
int counter = 0;  // Shared global variable
pthread_mutex_t lock;  // Mutex for protecting counter

void* increment(void* arg) {
    for (int i = 0; i < 100000; i++) {
        pthread_mutex_lock(&lock);     // Lock before accessing shared data
        counter++;
        pthread_mutex_unlock(&lock);   // Unlock after done
    }
    return NULL;
}

int main() {
    pthread_t t1, t2;

    pthread_mutex_init(&lock, NULL);  // Initialize the mutex

    pthread_create(&t1, NULL, increment, NULL);
    pthread_create(&t2, NULL, increment, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    pthread_mutex_destroy(&lock);  // Clean up

    printf("Final counter value: %d\n", counter);
    return 0;
}
```

Parallel Programming With `std::thread`

C++11 – std::threads

What is std::thread ?

- **`std::thread`** is `C++11` standard thread library
- Provides high-level type-safe, and RAII-friendly interface for multi threading
- Available on Unix-like Operating systems and **Microsoft Windows**.
- Cleaner syntax and better error handling.

C++11 - std::threads

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

void SayHello() {
    std::cout << "Hello, world!" << std::endl;
}

int main() {
    std::thread threads[16];

    for (int i = 0; i < 16; ++i) {
        threads[i] = std::thread(SayHello);
    }

    for (int i = 0; i < 16; ++i) {
        threads[i].join();
    }

    return 0;
}
```

Hello World Program

Compilation with g++

```
g++ -std=c++11 -pthread -o hello_threads hello_threads.cpp
```

Compilation with cmake

```
cmake_minimum_required(VERSION 3.10)
project(HelloThreads)

set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED True)

add_executable(hello_threads main.cpp)
```

C++11 – std::threads

Mutex initialization

Protecting Critical Section

Forking threads

Joining threads

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

int counter = 0;
std::mutex mtx;

void increment() {
    for (int i = 0; i < 100000; ++i) {
        std::lock_guard<std::mutex> lock(mtx);  // Scoped locking
        counter++;
    }
}

int main() {
    std::thread t1(increment);
    std::thread t2(increment);

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

    std::cout << "Final counter: " << counter << std::endl;
    return 0;
}
```

Summary of Programming with Threads

POSIX and C++ standard threads:

- Can be used from multiple languages
- Fine-grained control over threads (e.g. custom thread pools)
- Asynchronous or event-driven programming
- Popular in web servers, GUI, file I/O, OS level services, Message queues, etc.

Summary of Programming with Threads

POSIX and C++ standard threads:

- Can be used from multiple languages
- Fine-grained control over threads (e.g. custom thread pools)
- Asynchronous or event-driven programming
- Popular in web servers, GUI, file I/O, OS level services, Message queues, etc.

Pitfalls:

- Overhead of thread creation is high
- Data race bugs are very nasty to find
- Deadlocks are easier to find but can also be intermittent.
- Hard to parallelize loops (e.g. scientific applications).

Summary of Programming with Threads

POSIX and C++ standard threads:

- Can be used from multiple languages
- Fine-grained control over threads (e.g. custom thread pools)
- Asynchronous or event-driven programming
- Popular in web servers, GUI, file I/O, OS level services, Message queues, etc.

Pitfalls:

- Overhead of thread creation is high
- Data race bugs are very nasty to find
- Deadlocks are easier to find but can also be intermittent.
- Hard to parallelize loops (e.g. scientific applications).

OpenMP is commonly used today as an alternative, specially in applications such as scientific simulations, image and signal processing, matrix computations, machine learning, and etc.

Parallel Programming With `OpenMP`

What is OpenMP?

OpenMP stands for **Open** specification for **M**ulti-**P**rocessing

- https://openmp.org (community, talks, tutorials, etc.)
- A high-level **API** for parallel programming, mainly used in `C`, `C++`, and `Fortran`
  - **Preprocessor (compiler) directives (~ 80%)**
    - `#pragma omp construct[clause[clause …]]`
  - **Library Calls (~ 19%)**
    - `#include <omp.h>`
  - **Environment Variables (~ 1%)**

OpenMP

The OpenMP API specification for parallel programming

Home | Specifications | Community | Resources | News & Events | About

OpenMP ARB Releases the OpenMP API 6.0

- Improved tasking, device, induction, and base language support
- New Workdistribute directive and extended loop transformations

DOWNLOAD

Latest News and Events

ISC High Performance
June 10–13, 2025 | Hamburg, Germany
Attend the OpenMP Tutorials and BOF
ISC 2025

BLOG POST: OpenMP API 6.0 Implementations
Status of OpenMP API 6.0 Implementations

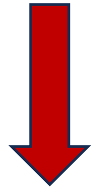
IWOMP 2025
UNC Charlotte, North Carolina, US
IWOMP 2025
IWOMP 2025 will be held at the UNC Charlotte

What is OpenMP from programmer's point of view?

OpenMP is a portable, threaded, shared memory programming specification with light syntax.

What is OpenMP from programmer's point of view?

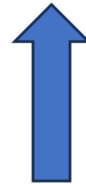
OpenMP is a portable, threaded, shared memory programming specification with light syntax.



Works across different operating systems and compilers

What is OpenMP from programmer's point of view?

Uses threads (POSIX threads) internally

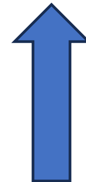


OpenMP is a portable, threaded, shared memory programming specification with light syntax.

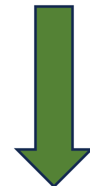
Works across different operating systems and compilers

What is OpenMP from programmer's point of view?

Uses threads (POSIX threads) internally



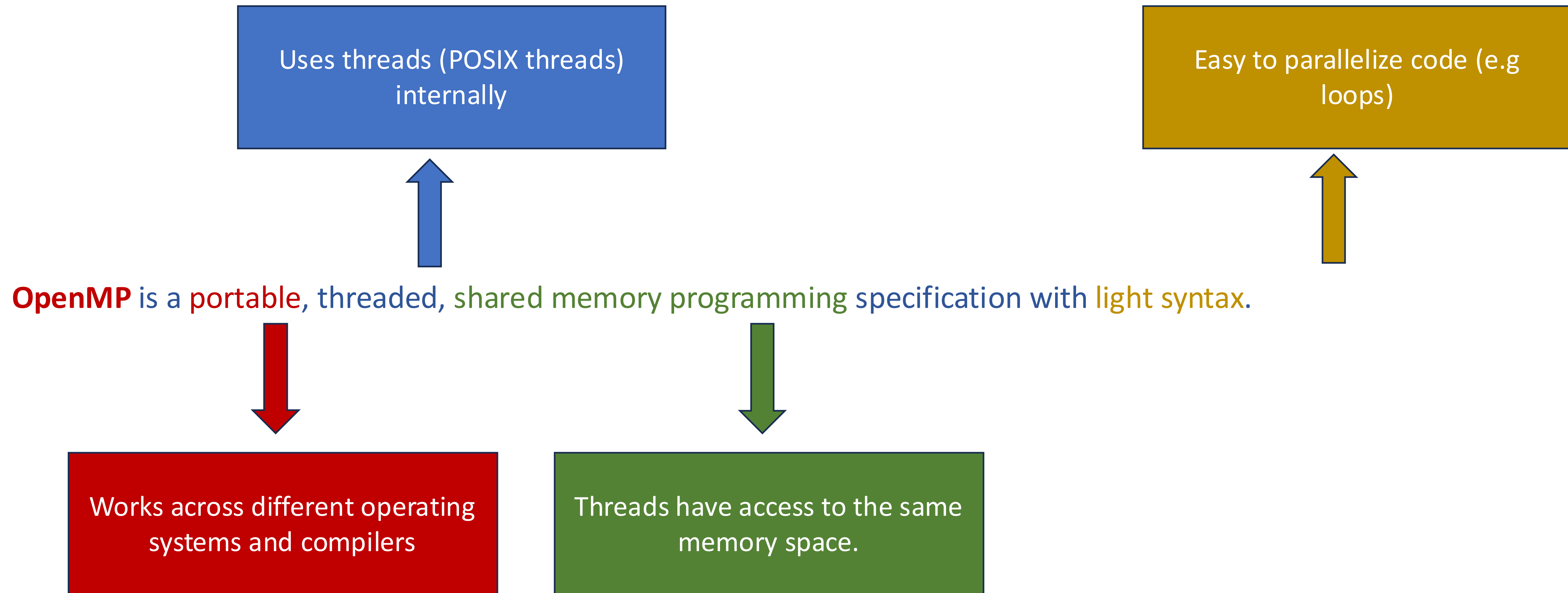
OpenMP is a portable, threaded, shared memory programming specification with light syntax.



Works across different operating systems and compilers

Threads have access to the same memory space.

What is OpenMP from programmer's point of view?



What is OpenMP from programmer's point of view?

OpenMP will

- **Allow a programmer to separate a program into serial regions and parallel regions.**
  - **Serial regions** - run by a single thread
  - **Parallel regions** – run by multiple threads
- **Hide stack management**
  - Thread creation
  - Stack size
  - Thread local storage
- **Provide synchronization constructs.**
  - Easy to use tools to prevent data races

What is OpenMP from programmer's point of view?

OpenMP will

- **Allow a programmer to separate a program into serial regions and parallel regions.**
  - **Serial regions** - run by a single thread
  - **Parallel regions** – run by multiple threads
- **Hide stack management**
  - Thread creation
  - Stack size
  - Thread local storage
- **Provide synchronization constructs.**
  - Easy to use tools to prevent data races

OpenMP will NOT

- **Parallelize automatically**
  - You must explicitly tell the compiler how to parallelize
- **Guarantee speedup**
  - Overheads from
    - Thread creation
    - Synchronization
- **Provide freedom from data races.**
  - OpenMP gives you tools, but it's programmer's responsibility to avoid data races.

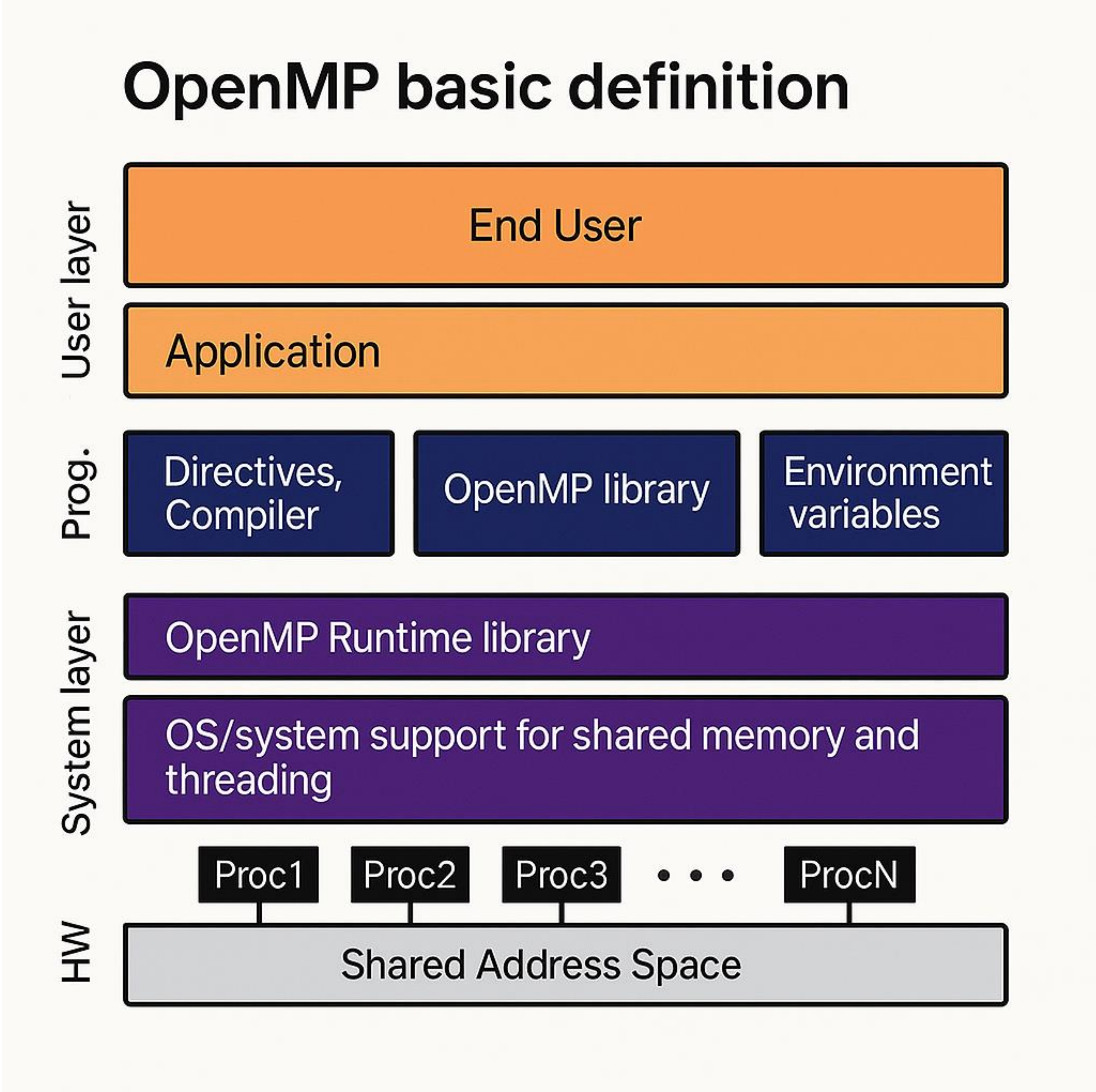
OpenMP popular directives

OpenMP pragma, function, or clause	Concepts
`#pragma omp parallel`	Creates a parallel region where a team of threads is spawned.
`int omp_get_thread_num()` `int omp_get_num_threads()`	Returns the ID of the current thread Returns the total number of threads in the current parallel region
`double omp_get_wtime()`	Returns wall-clock time, used to measure performance
`setenv OMP_NUM_THREADS  N`	Internal control variables. Setting the default number of threads with an environment variable
`#pragma omp barrier` `#pragma omp critical`	All threads wait at this point before continuing Only one thread at a time can enter this block
`#pragma omp for` `#pragma omp parallel for`	Used inside a parallel region to distribute loop iteration Shortcut: creates parallel region and distribute loop in one line.
`reduction(op:list)`	Reductions of values across a team of threads
`schedule(dynamic [,chunk])` `schedule (static [,chunk])`	Threads grab chunks as the finish Divides loop iterations evenly among threads

`private(list)`
`firstprivate(list)`
`shared(list)`
	Each thread gets its own uninitialized copy of variables in the list
	Each thread gets its own copy initialized with the value from the master thread
	Variables in the list are shared across all threads

`nowait`	Removes the implicit barrier at the end of a for, section, or single block
`#pragma omp single`	Code inside runs only by one thread, others skip it but wait
`#pragma omp task` `#pragma omp taskwait`	Creates a deferred unit of work (task) to be run by any thread Waits until all previously created tasks in this thread have completed.

OpenMP Architecture



Hello World!

```
#include <iostream>
#include <omp.h>

int main() {
    #pragma omp parallel
    {
        int tid = omp_get_thread_num();
        std::cout << "Hello, world! from thread " << tid << std::endl;
    }
    return 0;
}
```

Hello World program in OpenMP

Compilation with g++

```
g++ -fopenmp -o hello_openmp hello_openmp.cpp
```

Compilation with cmake

```
cmake_minimum_required(VERSION 3.30)
project(HelloOpenMP LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

find_package(OpenMP REQUIRED)
add_executable(hello_openmp hello_openmp.cpp)
target_link_libraries(hello_openmp PRIVATE OpenMP::OpenMP_CXX)
```

Hello World!

```cpp
#include <iostream>
#include <omp.h>

int main() {
    #pragma omp parallel
    {
        int tid = omp_get_thread_num();
        std::cout << "Hello, world! from thread " << tid << std::endl;
    }
    return 0;
}
```

Parallel Region

Hello World program in OpenMP

Compilation with g++

```
g++ -fopenmp -o hello_openmp hello_openmp.cpp
```

Compilation with cmake

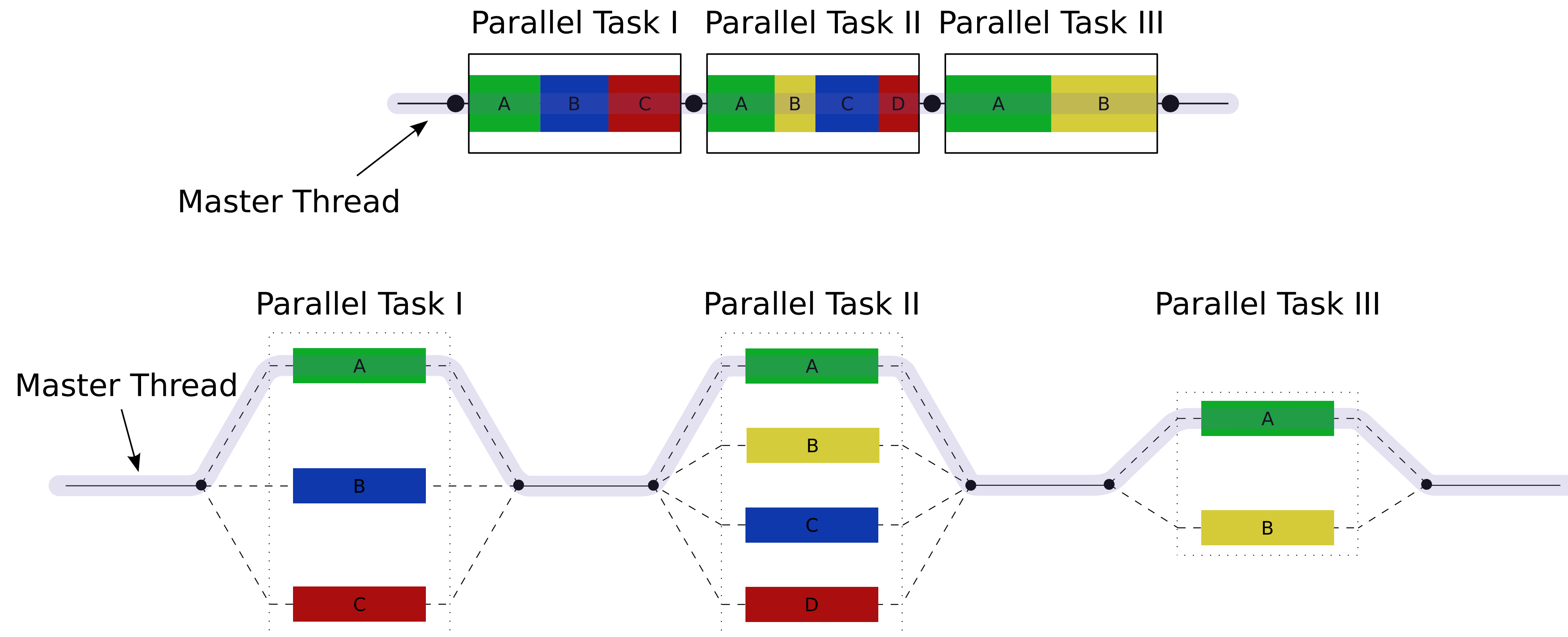
```
cmake_minimum_required(VERSION 3.30)
project(HelloOpenMP LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

find_package(OpenMP REQUIRED)
add_executable(hello_openmp hello_openmp.cpp)
target_link_libraries(hello_openmp PRIVATE OpenMP::OpenMP_CXX)
```

OpenMP programming model

Fork– Join Parallelism

Mater (main) thread spawns a team of threads as needed.

Parallelism added incrementally until the performance goals are met



Thread creation

- You create **a team of threads** in OpenMP with the parallel construct.
- You can request several threads with **omp_set_num_threads()**

- Is the number of requested threads the same as the number you actually get?

  - NO. An **implementation can silently decide to give you a team with fewer threads**.
  - OpenMP does not reduce the size of the team, once it's created

```
const int N = 1000;
std::vector<int> A(N, 1);
std::vector<int> B(N, 2);
std::vector<int> C(N);

omp_set_num_threads(4);

#pragma omp parallel for
for (int i = 0; i < N; ++i) {
    C[i] = A[i] + B[i];
}

for (int i = 0; i < 10; ++i) {
    std::cout << "C[" << i << "] = " << C[i] << std::endl;
}
```

Parallel vector addition program in OpenMP

Thread creation

- You create **a team of threads** in OpenMP with the parallel construct.
- You can request a number of threads with **`omp_set_num_threads()`**
- Is the number of requested threads the same as the number you actually get?
  - NO. An **implementation can silently decide to give you a team with fewer threads**.
  - OpenMP does not reduce the size of the team, once it's created

```
const int N = 1000;
std::vector<int> A(N, 1);
std::vector<int> B(N, 2);
std::vector<int> C(N);

omp_set_num_threads(4);

#pragma omp parallel for
for (int i = 0; i < N; ++i) {
    C[i] = A[i] + B[i];
}
```

Each thread executes a copy of the code within the block

Parallel vector addition program in OpenMP

The actual number of threads: `omp_get_num_threads()`

Within a parallel region

Next Lecture: OpenMP II + Parallel Numerical integration